

Susanne Albers  
Tomasz Radzik (Eds.)

LNCS 3221

# Algorithms – ESA 2004

12th Annual European Symposium  
Bergen, Norway, September 2004  
Proceedings



Springer

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

David Hutchison

*Lancaster University, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*University of Dortmund, Germany*

Madhu Sudan

*Massachusetts Institute of Technology, MA, USA*

Demetri Terzopoulos

*New York University, NY, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Moshe Y. Vardi

*Rice University, Houston, TX, USA*

Gerhard Weikum

*Max-Planck Institute of Computer Science, Saarbruecken, Germany*

Susanne Albers Tomasz Radzik (Eds.)

# Algorithms – ESA 2004

12th Annual European Symposium  
Bergen, Norway, September 14-17, 2004  
Proceedings



Springer

## Volume Editors

Susanne Albers  
Albert-Ludwigs-Universität Freiburg  
Institut für Informatik  
Georges-Köhler-Allee 79, 79110 Freiburg, Germany  
E-mail: salbers@informatik.uni-freiburg.de

Tomasz Radzik  
King's College London  
Department of Computer Science  
London WC2R 2LS, UK  
E-mail: radzik@dcs.kcl.ac.uk

Library of Congress Control Number: 2004111289

CR Subject Classification (1998): F.2, G.1-2, E.1, F.1.3, I.3.5, C.2.4, E.5

ISSN 0302-9743

ISBN 3-540-23025-4 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2004  
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Protago-TeX-Production GmbH  
Printed on acid-free paper SPIN: 11319627 06/3142 5 4 3 2 1 0

# Preface

This volume contains the 70 contributed papers and abstracts of two invited lectures presented at the 12th Annual European Symposium on Algorithms (ESA 2004), held in Bergen, Norway, September 14–17, 2004. The papers in each section of the proceedings are arranged alphabetically. The three distinguished invited speakers were David Eppstein, Michael Fellows and Monika Henzinger.

As in the last two years, ESA had two tracks, with separate program committees, which dealt respectively with:

- the design and mathematical analysis of algorithms (the “Design and Analysis” track);
- real-world applications, engineering and experimental analysis of algorithms (the “Engineering and Applications” track).

Previous ESAs were held at Bad Honnef, Germany (1993); Utrecht, The Netherlands (1994); Corfu, Greece (1995); Barcelona, Spain (1996); Graz, Austria (1997); Venice, Italy (1998); Prague, Czech Republic (1999); Saarbrücken, Germany (2000); Århus, Denmark (2001); Rome, Italy (2002); and Budapest, Hungary (2003). The predecessor to the Engineering and Applications track of ESA was the annual Workshop on Algorithm Engineering (WAE). Previous WAEs were held in Venice, Italy (1997); Saarbrücken, Germany (1998); London, UK (1999); Saarbrücken, Germany (2000); and Århus, Denmark (2001).

The proceedings of the previous ESAs were published as Springer-Verlag’s LNCS volumes 726, 855, 979, 1284, 1461, 1643, 1879, 2161, 2461, and 2832. The proceedings of the WAEs from 1999 onwards were published as Springer-Verlag’s LNCS volumes 1668, 1982, and 2141.

Papers were solicited in all areas of algorithmic research, including but not limited to: computational biology, computational finance, computational geometry, databases and information retrieval, external-memory algorithms, graph and network algorithms, graph drawing, machine learning, network design, online algorithms, parallel and distributed computing, pattern matching and data compression, quantum computing, randomized algorithms, and symbolic computation. The algorithms could be sequential, distributed, or parallel. Submissions were strongly encouraged in the areas of mathematical programming and operations research, including: approximation algorithms, branch-and-cut algorithms, combinatorial optimization, integer programming, network optimization, polyhedral combinatorics, and semidefinite programming.

Each extended abstract was submitted to exactly one of the two tracks, and a few abstracts were switched from one track to the other at the discretion of the program chairs during the reviewing process. The extended abstracts were read by at least four referees each, and evaluated on their quality, originality, and relevance to the symposium. The program committees of both tracks met at King’s College London at the end of May. The Design and Analysis track selected for presentation 52 out of 158 submitted abstracts. The Engineering and

Applications track selected for presentation 18 out of 50 submitted abstracts. The program committees of the two tracks consisted of:

#### Design and Analysis Track

Karen Aardal	(CWI, Amsterdam)
Susanne Albers (Chair)	(University of Freiburg)
Timothy Chan	(University of Waterloo)
Camil Demetrescu	(University of Rome “La Sapienza”)
Rolf Fagerberg	(BRICS Århus and University of Southern Denmark)
Paolo Ferragina	(University of Pisa)
Fedor Fomin	(University of Bergen)
Claire Kenyon	(École Polytechnique, Palaiseau)
Elias Koutsoupias	(University of Athens and UCLA)
Klaus Jansen	(University of Kiel)
Ulrich Meyer	(MPI Saarbrücken)
Michael Mitzenmacher	(Harvard University)
Joseph (Seffi) Naor	(Technion, Haifa)
Micha Sharir	(Tel Aviv University)
Peter Widmayer	(ETH Zürich)
Gerhard Woeginger	(University of Twente and TU Eindhoven)

#### Engineering and Applications Track

Bo Chen	(University of Warwick)
Ulrich Derigs	(University of Cologne)
Andrew V. Goldberg	(Microsoft Research, Mountain View)
Roberto Grossi	(University of Pisa)
Giuseppe F. Italiano	(University of Rome “Tor Vergata”)
Giuseppe Liotta	(University of Perugia)
Tomasz Radzik (Chair)	(King’s College London)
Marie-France Sagot	(INRIA Rhône-Alpes)
Christian Scheideler	(Johns Hopkins University)
Jop F. Sibeyn	(University of Halle)
Michiel Smid	(Carleton University)

ESA 2004 was held along with the 4th Workshop on Algorithms in Bioinformatics (WABI 2004), the 4th Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS 2004), the 2nd Workshop on Approximation and Online Algorithms (WAOA 2004), and an International Workshop on Parametrized and Exact Computation (IWPEC 2004) in the context of the combined conference ALGO 2004. The organizing committee of ALGO 2004 consisted of Pinar Heggernes (chair), Fedor Fomin (co-chair), Eivind Coward, Inge Jonassen, Fredrik Manne, and Jan Arne Telle, all from the University of Bergen.

ESA 2004 was sponsored by EATCS (the European Association for Theoretical Computer Science), the University of Bergen, the Research Council of

Norway, IBM, HP and SGI. The EATCS sponsorship included an award of EUR 500 for the authors of the best student paper at ESA 2004. The winners of the prize were Marcin Muca and Piotr Sandowski for their paper *Maximum Matching in Planar Graphs via Gaussian Elimination*.

Finally, we would like to thank the members of the program committees for their time and work devoted to the paper selection process.

July 2004

Susanne Albers and Tomasz Radzik  
Program Chairs, ESA 2004

## Referees

Mohamed I. Abouelhoda	Gianna Del Corso
Pankaj Kumar Agarwal	Erik Demaine
Noga Alon	Roman Dementiev
Steven Alpern	Jörg Derungs
Ernst Althaus	Tamal Dey
Christoph Ambühl	Emilio Di Giacomo
Luzi Anderegg	Walter Didimo
Lars Arge	Martin Dietzfelbinger
Vera Asodi	Krzysztof Diks
Franz Aurenhammer	Yefim Dinitz
Yossi Azar	Florian Dittrich
Holger Bast	Debora Donato
Surender Baswana	Eleni Drinea
Luca Becchetti	Vida Dujmovic
Rene Beier	Christian Duncan
Michael Bender	Stefan Edelkamp
Marc Benkert	Herbert Edelsbrunner
Mark de Berg	Alon Efrat
Randeep Bhatia	Stephan Eidenbenz
Ulrik Brandes	Lars Engebretsen
Gerth Stølting Brodal	Roe Engelberg
Hajo Broersma	David Eppstein
Adam Buchsbaum	Leah Epstein
Stefan Burkhardt	Jeff Erickson
Gruia Calinescu	Thomas Erlebach
Saverio Caminiti	Lene Monrad Favrholdt
Frédéric Cazals	Sandor Fekete
Bernard Chazelle	Mike Fellows
Chandra Chekuri	Amos Fiat
Johnny Chen	Irene Finocchi
Joseph Cheriyan	Aleksei Fishkin
Steve Chien	Lisa Fleischer
Jana Chlebikova	Rudolf Fleischer
Marek Chrobak	Paola Flocchini
Julia Chuzhoy	Pierre Fraigniaud
Mark Cieliebak	Gianni Franceschini
Valentina Ciriani	Paolo Franciosa
Andrea Clementi	Gudmund Skovbjerg Frandsen
Richard Cole	Antonio Frangioni
Colin Cooper	Ari Freund
Andreas Crauser	Daniele Frigioni
Janos Csirik	Stefan Funke
Victor Dalmau	Martin Fürer
Siavash Vahdati Daneshmand	Hal Gabow



Bernd Gärtner  
Naveen Garg  
Leszek Gasieniec  
Olga Gerber  
Georg Gottlob  
Fabrizio Grandoni  
Roberto Grossi  
Jens Gustedt  
Magnús Halldórsson  
Dan Halperin  
Sariel Har-Peled  
David Hart  
Jason Hartline  
Refael Hassin  
Ryan Hayward  
Danny Hermelin  
Volker Heun  
Stefan Hougardy  
Cor Hurkens  
Sandy Irani  
Rob W. Irving  
Riko Jacob  
Tommy Jensen  
Frank Kammer  
Viggo Kann  
Haim Kaplan  
Juha Kärkkäinen  
Brad Karp  
Irit Katriel  
Michael Kaufmann  
Hans Kellerer  
Lutz Kettner  
Valerie King  
Tamas Kiraly  
Adam Kirsch  
Teemu Kivioja  
Ralf Klasing  
Bettina Klinz  
Daniel Kobler  
Jochen Konemann  
Alexandr Kononov  
Michael Korn  
Guy Kortsarz  
Dariusz Kowalski  
Daniel Kral  
Marc van Kreveld  
Michael Krivelevich  
Piotr Krysta  
Alejandro López-Ortiz  
Gad Landau  
Kim Skak Larsen  
James Lee  
Gary G. Lee  
Moshe Lewenstein  
Liane Lewin-Eytan  
Andrzej Lingas  
Dean Lorenz  
Anna Lubiw  
Fabrizio Luccio  
Flaminia Luccio  
Tamas Lukovski  
Anil Maheshwari  
Thomas Mailund  
Christos Makris  
Ion Mandoiu  
David Manlove  
Fredrik Manne  
Giovanni Manzini  
Gitta Marchand  
Alberto Marchetti-Spaccamela  
David Orden Martin  
Jirka Matousek  
Ernst W. Mayr  
Alessandro Mei  
Peter Bro Miltersen  
Joseph Mitchell  
Anders Møller  
Angelo Monti  
Pat Morin  
Aziz Moukrim  
Haiko Müller  
S. Muthukrishnan  
Gene Myers  
Umberto Nanni  
Morten Hegner Nielsen  
Naomi Nishimura  
Marc Nunkesser  
Liadan O'Callaghan  
Anna Östlin Pagh  
Rasmus Pagh

Linda Pagli  
 Aris Pagourtzis  
 Igor Pak  
 Anna Palbom  
 Alessandro Panconesi  
 Gopal Pandurangan  
 Maurizio Patrignani  
 Marcin Peczarski  
 Christian N.S. Pedersen  
 Leon Peeters  
 Marco Pellegrini  
 Rudi Pendavingh  
 Paolo Penna  
 Giuseppe Persiano  
 Seth Pettie  
 Andrea Pietracaprina  
 Maurizio Pizzonia  
 Greg Plaxton  
 Andrzej Proskurowski  
 Kirk Pruhs  
 Geppino Pucci  
 Uri Rabinovich  
 Mathieu Raffinot  
 Sven Rahmann  
 Rajmohan Rajaraman  
 Edgar Ramos  
 Michael Rao  
 Srinivasa Rao  
 R. Ravi  
 Dror Rawitz  
 Bruce Reed  
 Oded Regev  
 Franz Rendl  
 Romeo Rizzi  
 Liam Roditty  
 Hein Röhrig  
 Günter Rote  
 Tim Roughgarden  
 Amin Saberi  
 Cenk Sahinalp  
 Jared Saia  
 Peter Sanders  
 Miklos Santha  
 Nicola Santoro  
 Fabiano Sarracco

Joe Sawada  
 Cynthia Sawchuk  
 Gabriel Scalosub  
 Guido Schäfer  
 Baruch Schieber  
 Manfred Schimmler  
 Christian Schindelhauer  
 Klaus Schittkowski  
 Susanne Schmitt  
 Frank Schulz  
 Maria Serna  
 Jiri Sgall  
 Ron Shamir  
 Roded Sharan  
 Bruce Shepherd  
 David Shmoys  
 Riccardo Silvestri  
 Amitabh Sinha  
 Naveen Sivadasan  
 Martin Skutella  
 Roberto Solis-Oba  
 Bettina Speckmann  
 Frits Spiessma  
 Divesh Srivastava  
 Rob van Stee  
 Cliff Stein  
 Ileana Streinu  
 Maxim Sviridenko  
 Gabor Szabo  
 Arie Tamir  
 Eric Tannier  
 Jan Arne Telle  
 Moshe Tennenholtz  
 Thorsten Theobald  
 Dimitrios Thilikos  
 Ralf Thöle  
 Torsten Tholey  
 Carsten Thomassen  
 Mikkel Thorup  
 Ioan Todinca  
 Laura Toma  
 Esko Ukkonen  
 Peter Ullrich  
 Takeaki Uno  
 Eli Upfal

Berthold Vöcking  
Jan Vahrenhold  
Kasturi Varadarajan  
Elad Verbin  
Stephane Vialette  
Eric Vigoda  
Luca Vismara  
Tjark Vredeveld  
Mirjam Wattenhofer  
Birgitta Weber  
Michael Weber  
Udi Wieder  
Ryan Williams

Alexander Wolff  
David Wood  
Deshi Ye  
Anders Yeo  
Neal Young  
W. Zang  
Christos Zaroliagis  
Norbert Zeh  
Alex Zelikovski  
Guochuan Zhang  
Hu Zhang  
Uri Zwick

# Table of Contents

## Invited Lectures

A Survey of FPT Algorithm Design Techniques with an Emphasis on Recent Advances and Connections to Practical Computing . . . . .	1
<i>Michael R. Fellows</i>	
Algorithmic Aspects of Web Search Engines . . . . .	3
<i>Monika Henzinger</i>	

## Design and Analysis Track

Efficient Tradeoff Schemes in Data Structures for Querying Moving Objects . . . . .	4
<i>Pankaj K. Agarwal, Lars Arge, Jeff Erickson, Hai Yu</i>	
Swap and Mismatch Edit Distance . . . . .	16
<i>Amihoud Amir, Estrella Eisenberg, Ely Porat</i>	
Path Decomposition Under a New Cost Measure with Applications to Optical Network Design . . . . .	28
<i>Elliot Anshelevich, Lisa Zhang</i>	
Optimal External Memory Planar Point Enclosure . . . . .	40
<i>Lars Arge, Vasilis Samoladas, Ke Yi</i>	
Maximizing Throughput in Multi-queue Switches . . . . .	53
<i>Yossi Azar, Arik Litichevsky</i>	
An Improved Algorithm for CIOQ Switches . . . . .	65
<i>Yossi Azar, Yossi Richter</i>	
Labeling Smart Dust . . . . .	77
<i>Vikas Bansal, Friedhelm Meyer auf der Heide, Christian Sohler</i>	
Graph Decomposition Lemmas and Their Role in Metric Embedding Methods . . . . .	89
<i>Yair Bartal</i>	
Modeling Locality: A Probabilistic Analysis of LRU and FWF . . . . .	98
<i>Luca Becchetti</i>	
An Algorithm for Computing DNA Walks . . . . .	110
<i>Ankur Bhargava, S. Rao Kosaraju</i>	

Algorithms for Generating Minimal Blockers of Perfect Matchings in Bipartite Graphs and Related Problems .....	122
<i>Endre Boros, Khaled Elbassioni, Vladimir Gurvich</i>	
Direct Routing: Algorithms and Complexity .....	134
<i>Costas Busch, Malik Magdon-Ismail, Marios Mavronicolas, Paul Spirakis</i>	
Lower Bounds for Embedding into Distributions over Excluded Minor Graph Families .....	146
<i>Douglas E. Carroll, Ashish Goel</i>	
A Parameterized Algorithm for Upward Planarity Testing .....	157
<i>Hubert Chan</i>	
Fisher Equilibrium Price with a Class of Concave Utility Functions .....	169
<i>Ning Chen, Xiaotie Deng, Xiaoming Sun, Andrew Chi-Chih Yao</i>	
Hardness and Approximation Results for Packing Steiner Trees .....	180
<i>Joseph Cheriyan, Mohammad R. Salavatipour</i>	
Approximation Hardness of Dominating Set Problems .....	192
<i>Miroslav Chlebík, Janka Chlebíková</i>	
Improved Online Algorithms for Buffer Management in QoS Switches ....	204
<i>Marek Chrobak, Wojciech Jawor, Jiří Sgall, Tomáš Tichý</i>	
Time Dependent Multi Scheduling of Multicast .....	216
<i>Rami Cohen, Dror Rawitz, Danny Raz</i>	
Convergence Properties of the Gravitational Algorithm in Asynchronous Robot Systems .....	228
<i>Reuven Cohen, David Peleg</i>	
The Average Case Analysis of Partition Sorts .....	240
<i>Richard Cole, David C. Kandathil</i>	
A Fast Distributed Algorithm for Approximating the Maximum Matching .....	252
<i>Andrzej Czygrinow, Michał Hańćkowiak, Edyta Szymańska</i>	
Extreme Points Under Random Noise .....	264
<i>Valentina Damerow, Christian Sohler</i>	
Fixed Parameter Algorithms for Counting and Deciding Bounded Restrictive List $H$ -Colorings .....	275
<i>Josep Díaz, Maria Serna, Dimitrios M. Thilikos</i>	
On Variable-Sized Multidimensional Packing .....	287
<i>Leah Epstein, Rob van Stee</i>	

An Inductive Construction for Plane Laman Graphs via Vertex Splitting .....	299
<i>Zsolt Fekete, Tibor Jordán, Walter Whiteley</i>	
Faster Fixed-Parameter Tractable Algorithms for Matching and Packing Problems .....	311
<i>Michael R. Fellows, C. Knauer, N. Nishimura, P. Ragde, F. Rosamond, U. Stege, Dimitrios M. Thilikos, S. Whitesides</i>	
On the Evolution of Selfish Routing .....	323
<i>Simon Fischer, Berthold Vöcking</i>	
Competitive Online Approximation of the Optimal Search Ratio .....	335
<i>Rudolf Fleischer, Tom Kamphans, Rolf Klein, Elmar Langetepe, Gerhard Trippen</i>	
Incremental Algorithms for Facility Location and $k$ -Median .....	347
<i>Dimitris Fotakis</i>	
Dynamic Shannon Coding .....	359
<i>Travis Gagie</i>	
Fractional Covering with Upper Bounds on the Variables: Solving LPs with Negative Entries .....	371
<i>Naveen Garg, Rohit Khandekar</i>	
Negotiation-Range Mechanisms: Coalition-Resistant Markets .....	383
<i>Rica Gonen</i>	
Approximation Algorithms for Quickest Spanning Tree Problems .....	395
<i>Refael Hassin, Asaf Levin</i>	
An Approximation Algorithm for Maximum Triangle Packing .....	403
<i>Refael Hassin, Shlomi Rubinstein</i>	
Approximate Parameterized Matching .....	414
<i>Carmit Hazay, Moshe Lewenstein, Dina Sokol</i>	
Approximation of Rectangle Stabbing and Interval Stabbing Problems ...	426
<i>Sofia Kovaleva, Frits C.R. Spijksma</i>	
Fast 3-Coloring Triangle-Free Planar Graphs .....	436
<i>Lukasz Kowalik</i>	
Approximate Unions of Lines and Minkowski Sums .....	448
<i>Marc van Kreveld, A. Frank van der Stappen</i>	
Radio Network Clustering from Scratch .....	460
<i>Fabian Kuhn, Thomas Moscibroda, Roger Wattenhofer</i>	

Seeking a Vertex of the Planar Matching Polytope in NC .....	472
<i>Raghav Kulkarni, Meena Mahajan</i>	
Equivalence of Search Capability Among Mobile Guards with Various Visibilities.....	484
<i>Jae-Ha Lee, Sang-Min Park, Kyung-Yong Chwa</i>	
Load Balancing in Hypercubic Distributed Hash Tables with Heterogeneous Processors.....	496
<i>Junning Liu, Micah Adler</i>	
On the Stability of Multiple Partner Stable Marriages with Ties .....	508
<i>Varun S. Malhotra</i>	
Flows on Few Paths: Algorithms and Lower Bounds .....	520
<i>Maren Martens, Martin Skutella</i>	
Maximum Matchings in Planar Graphs via Gaussian Elimination .....	532
<i>Marcin Mucha, Piotr Sankowski</i>	
Fast Multipoint Evaluation of Bivariate Polynomials.....	544
<i>Michael Nüsken, Martin Ziegler</i>	
On Adaptive Integer Sorting .....	556
<i>Anna Pagh, Rasmus Pagh, Mikkel Thorup</i>	
Tiling a Polygon with Two Kinds of Rectangles .....	568
<i>Eric Rémila</i>	
On Dynamic Shortest Paths Problems .....	580
<i>Liam Roditty, Uri Zwick</i>	
Uniform Algorithms for Deterministic Construction of Efficient Dictionaries .....	592
<i>Milan Ružić</i>	
Fast Sparse Matrix Multiplication .....	604
<i>Raphael Yuster, Uri Zwick</i>	

## Engineering and Applications Track

An Experimental Study of Random Knapsack Problems .....	616
<i>Rene Beier, Berthold Vöcking</i>	
Contraction and Treewidth Lower Bounds.....	628
<i>Hans L. Bodlaender, Arie M.C.A. Koster, Thomas Wolle</i>	
Load Balancing of Indivisible Unit Size Tokens in Dynamic and Heterogeneous Networks .....	640
<i>Robert Elsässer, Burkhard Monien, Stefan Schamberger</i>	

Comparing Real Algebraic Numbers of Small Degree .....	652
<i>Ioannis Z. Emiris, Elias P. Tsigaridas</i>	
Code Flexibility and Program Efficiency by Genericity: Improving CGAL's Arrangements .....	664
<i>Efi Fogel, Ron Wein, Dan Halperin</i>	
Finding Dominators in Practice .....	677
<i>Loukas Georgiadis, Renato F. Werneck, Robert E. Tarjan, Spyridon Triantafyllis, David I. August</i>	
Data Migration on Parallel Disks .....	689
<i>Leana Golubchik, Samir Khuller, Yoo-Ah Kim, Svetlana Shargorodskaya, Yung-Chun (Justin) Wan</i>	
Classroom Examples of Robustness Problems in Geometric Computations .....	702
<i>Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, Chee Yap</i>	
Stable Minimum Storage Merging by Symmetric Comparisons .....	714
<i>Pok-Son Kim, Arne Kutzner</i>	
On Rectangular Cartograms .....	724
<i>Marc van Kreveld, Bettina Speckmann</i>	
Multi-word Atomic Read/Write Registers on Multiprocessor Systems ....	736
<i>Andreas Larsson, Anders Gidenstam, Phuong H. Ha, Marina Papatriantaofilou, Philippos Tsigas</i>	
Beyond Optimal Play in Two-Person-Zerosum Games .....	749
<i>Ulf Lorenz</i>	
Solving Geometric Covering Problems by Data Reduction .....	760
<i>Steffen Mecke, Dorothea Wagner</i>	
Efficient IP Table Lookup via Adaptive Stratified Trees with Selective Reconstructions .....	772
<i>Marco Pellegrini, Giordano Fusco</i>	
Super Scalar Sample Sort .....	784
<i>Peter Sanders, Sebastian Winkel</i>	
Construction of Minimum-Weight Spanners .....	797
<i>Mikkel Sigurd, Martin Zachariasen</i>	



A Straight Skeleton Approximating the Medial Axis ..... 809  
    *Mirela Tănase, Remco C. Veltkamp*

Non-additive Shortest Paths ..... 822  
    *George Tsaggouris, Christos Zaroliagis*

**Author Index** ..... 835

# A Survey of FPT Algorithm Design Techniques with an Emphasis on Recent Advances and Connections to Practical Computing

Michael R. Fellows

School of Electrical Engineering and Computer Science  
University of Newcastle, University Drive, Callaghan NSW 2308, Australia  
`mfellows@cs.newcastle.edu.au`

**Abstract.** The talk will survey the rich toolkit of FPT algorithm design methodologies that has developed over the last 20 years, including “older” techniques such as well-quasi-ordering, bounded treewidth, color-coding, reduction to a problem kernel, and bounded search trees — which have continued to deepen and advance — as well as some recently developed approaches such as win/win’s, greedy localization, iterative compression, and crown decompositions.

Only in the last few years has it become clear that there are two distinct theoretical “games” being played in the effort to devise better and better FPT algorithms for various problems, that is, algorithms with a running time of  $O(f(k)n^c)$ , where  $c$  is a fixed constant,  $k$  is the parameter, and  $n$  is the total input size. The first of these games is the obvious one of improving the parameter cost function  $f(k)$ . For example, the first FPT algorithm for VERTEX COVER had a running time of  $2^{kn}$ . After a series of efforts, the best currently known algorithm (due to Chandran and Grandoni, in a paper to be presented at IWPEC 2004), has a parameter cost function of  $f(k) = (1.2745)^k k^4$ . The second theoretical game being played in FPT algorithm design has to do with efficient kernelization. The naturality and universality of this game comes from the observation that a parameterized problem is FPT if and only if there is a *polynomial time* preprocessing (also called *data reduction* or *kernelization*) algorithm that transforms the input  $(x, k)$  into  $(x', k')$ , where  $k' \leq k$  and  $|x'| \leq g(k)$  for some kernel-bounding function  $g(k)$ , and where of course  $(x, k)$  is a yes-instance if and only if  $(x', k')$  is a yes-instance. In other words, modulo polynomial-time pre-processing, all of the difficulty, and even the size of the input that needs to be considered, is bounded by a function of the parameter. The natural theoretical game from this point of view about FPT is to improve the kernel-bounding function  $g(k)$ . For example, after strenuous efforts, a kernel-bounding function of  $g(k) = 335k$  has recently been achieved (by Alber et al.) for the PLANAR DOMINATING SET problem.

Some of the new techniques to be surveyed pertain to the  $f(k)$  game, some address the kernelization game, and some are relevant to both of these goals in improving FPT algorithms. One of the striking things about the “positive” toolkit of FPT algorithm design is how unsettled the area seems to be, with

seemingly rather basic methodological insights still emerging in recent years with surprising frequency. In other words, despite 20 years of research on FPT algorithm design, the area still seems to be in its infancy.

Most of the work on FPT algorithm design to date has been theoretical, sometimes highly theoretical and seemingly completely impractical (such as in FPT results obtained by well-quasiordering methods). The last few years have seen two developments concerning the practical relevance of FPT algorithmics.

First of all, there has been a surge of implementation efforts that have helped to clarify the somewhat complex relationship between the theoretical games and practical algorithms. In many cases, adaptations of the structural and algorithmic insights of the theoretical FPT results are yielding the best available practical algorithms for problems such as VERTEX COVER, regardless of whether the parameter is small or not! For example, polynomial-time pre-processing rules uncovered by the kernelization game will *always* be useful, and are sometimes of substantial commercial interest. Similarly, clever branching strategies that are often a key part of advances in the  $f(k)$  game are of universal significance with respect to backtracking heuristics for hard problems. Some striking examples of this interplay of FPT theory and practical implementations will be described by Langston in his IWPEC 2004 invited address.

Secondly, a broad realization has developed that many implemented practical algorithms for hard problems, that people are quite happy with, are actually FPT algorithms, unanalyzed and not described as such, sometimes even with quite terrible  $f(k)$ 's (if viewed theoretically) and employing relatively unsophisticated pre-processing. So it does seem that there is plenty of scope for theoretical research on improved FPT algorithms to have considerable impact on practical computing.

# Algorithmic Aspects of Web Search Engines

Monika Henzinger

Google Inc.  
Mountain View, CA 94043, USA  
`monika@google.com`

**Abstract.** Web search engines have emerged as one of the central applications on the internet. In fact, search has become one of the most important activities that people engage in on the Internet. Even beyond becoming the number one source of information, a growing number of businesses are depending on web search engines for customer acquisition. In this talk I will brief review the history of web search engines: The first generation of web search engines used text-only retrieval techniques. Google revolutionized the field by deploying the PageRank technology - an eigenvector-based analysis of the hyperlink structure- to analyze the web in order to produce relevant results. Moving forward, our goal is to achieve a better understanding of a page with a view towards producing even more relevant results.

Google is powered by a large number of PCs. Using this infrastructure and striving to be as efficient as possible poses challenging systems problems but also various algorithmic challenges. I will discuss some of them in my talk.

# Efficient Tradeoff Schemes in Data Structures for Querying Moving Objects<sup>\*</sup>

Pankaj K. Agarwal<sup>1</sup>, Lars Arge<sup>1</sup>, Jeff Erickson<sup>2</sup>, and Hai Yu<sup>1</sup>

<sup>1</sup> Department of Computer Science, Duke University,  
Durham, NC 27708-0129  
{`pankaj,large,fishhai`}@cs.duke.edu

<sup>2</sup> Department of Computer Science, University of Illinois,  
Urbana, IL 61801-2302  
`jeffe@cs.uiuc.edu`

**Abstract.** The ability to represent and query continuously moving objects is important in many applications of spatio-temporal database systems. In this paper we develop data structures for answering various queries on moving objects, including range and proximity queries, and study tradeoffs between various performance measures—query time, data structure size, and accuracy of results.

## 1 Introduction

With the rapid advances in geographical positioning technologies, it has become feasible to track continuously moving objects accurately. In many applications, one would like to store these moving objects so that various queries on them can be answered quickly. For example, a mobile phone service provider may wish to know how many users are currently present in a specified area. Unfortunately, most traditional data structure techniques assume that data do not change over time. Therefore the problem of efficiently representing and querying moving objects has been extensively researched in both the database and computational geometry communities. Recently, several new techniques have been developed; see [1,7,9,15,17,22] and references therein.

The kinetic data structure framework (KDS) proposed by Basch *et al.* [9] has been successfully applied to a variety of geometric problems to efficiently maintain data structures for moving objects. The key observation of KDS is that though the actual structure defined by a set of moving objects may change continuously over time, its combinatorial description changes only at discrete time instances. Although this framework has proven to be very useful for maintaining

---

<sup>\*</sup> Research by the first and fourth authors is supported by NSF under grants CCR-0086013, EIA-9870724, EIA-0131905, and CCR-0204118, and by a grant from the U.S.–Israel Binational Science Foundation. Research by the second author is supported by NSF under grants EIA-9972879, CCR-9984099, EIA-0112849, and INT-0129182. Research by the third author is supported by NSF under grants CCR-0093348, DMR-0121695 and CCR-0219594.

geometric structures, it is not clear whether it is the right framework if one simply wants to query moving objects: on the one hand, there is no need to maintain the underlying structure explicitly, and on the other hand the KDS maintains a geometric structure on the current configuration of objects, while one may actually wish to answer queries on the past as well as the future configurations of objects.

For example, one can use the KDS to maintain the convex hull of a set of linearly moving points [9]. While the points move,  $O(n^2)$  events are processed when the combinatorial description of the convex hull changes, each requiring  $O(\log^2 n)$  time. Using the maintained structure, one can easily determine whether a query point lies in the convex hull of the current configuration of points. But suppose one is interested in queries of the following form: “Does a point  $p$  lie in the convex hull of the point set at a future time  $t_q$ ?” In order to answer this query, there is no need to maintain the convex hull explicitly. The focus of this paper is to develop data structures for answering this and similar types of queries and to study tradeoffs between various performance measures such as query time, data structure size, and accuracy of results.

**Problem statement.** Let  $S = \{p_1, p_2, \dots, p_n\}$  be a set of moving points in  $\mathbb{R}^1$  or  $\mathbb{R}^2$ . For any time  $t$ , let  $p_i(t)$  be the position of  $p_i$  at time  $t$ , and let  $S(t) = \{p_1(t), p_2(t), \dots, p_n(t)\}$  be the configuration of  $S$  at time  $t$ . We assume that each  $p_i$  moves along a straight line at some fixed speed. We are interested in answering the following queries.

- **Interval query:** Given an interval  $I = [y_1, y_2] \subseteq \mathbb{R}^1$  and a time stamp  $t_q$ , report  $S(t_q) \cap I$ , that is, all  $k$  points of  $S$  that lie inside  $I$  at time  $t_q$ .
- **Rectangle query:** Given an axis-aligned rectangle  $R \subseteq \mathbb{R}^2$  and a time stamp  $t_q$ , report  $S(t_q) \cap R$ , that is, all  $k$  points of  $S$  that lie inside  $R$  at time  $t_q$ .
- **Approximate nearest-neighbor query:** Given a query point  $p \in \mathbb{R}^2$ , a time stamp  $t_q$  and a parameter  $\delta > 0$ , report a point  $p' \in S$  such that  $d(p, p'(t_q)) \leq (1 + \delta) \cdot \min_{q \in S} d(p, q(t_q))$ .
- **Approximate farthest-neighbor query:** Given a query point  $p \in \mathbb{R}^2$ , a time stamp  $t_q$ , and a parameter  $0 < \delta < 1$ , report a point  $p' \in S$  such that  $d(p, p'(t_q)) \geq (1 - \delta) \cdot \max_{q \in S} d(p, q(t_q))$ .
- **Convex-hull query:** Given a query point  $p \in \mathbb{R}^2$  and a time stamp  $t_q$ , determine whether  $p$  lies inside the convex hull of  $S(t_q)$ .

**Previous results.** Range trees and  $kd$ -trees are two widely used data structures for orthogonal range queries; both of these structures have been generalized to the kinetic setting. Agarwal *et al.* [1] developed a two-dimensional kinetic range tree that requires  $O(n \log n / \log \log n)$  space and answers queries in  $O(\log n + k)$  time.<sup>1</sup> Agarwal *et al.* [4] developed kinetic data structures for two variants of the

<sup>1</sup> Some of the previous work described in this section was actually done in the standard two-level I/O model aiming to minimize the number of I/Os. Here we interpret and state their results in the standard internal-memory setting.

standard  $kd$ -tree that can answer queries in  $O(n^{1/2+\varepsilon} + k)$  time. These kinetic data structures can only answer queries about the current configuration of points, that is, where  $t_q$  is the current time.

Kollios *et al.* [17] proposed a linear-size data structure based on partition trees [18] for answering interval queries with arbitrary time stamps  $t_q$ , in time  $O(n^{1/2+\varepsilon} + k)$ . This result was later extended to  $\mathbb{R}^2$  by Agarwal *et al.* [1] with the same space and query time bounds.

Agarwal *et al.* [1] were the first to propose *time-responsive* data structures, which answer queries about the current configuration of points more quickly than queries about the distant past or future. Agarwal *et al.* [1] developed a time-responsive data structure for interval and rectangle queries where the cost of any query is a monotonically increasing function of the difference between the query's *time stamp*  $t_q$  and the current time; the worst-case query cost is  $O(n^{1/2+\varepsilon} + k)$ . However, no exact bounds on the query time were known except for a few special cases.

Fairly complicated time-responsive data structures with provable query time bounds were developed later in [2]. The query times for these structures are expressed in terms of *kinetic events*; a kinetic event occurs whenever two moving points in  $S$  momentarily share the same  $x$ -coordinate or the same  $y$ -coordinate. For a query with time stamp  $t_q$ , let  $\varphi(t_q)$  denote the number kinetic events between  $t_q$  and the current time. The data structure uses  $O(n \log n)$  space, answers interval queries in  $O(\varphi(t_q)/n + \log n + k)$  time, and answers rectangle queries in  $O(\sqrt{\varphi(t_q)} + (\sqrt{n}/\sqrt{\lceil \varphi(t_q)/n \rceil}) \log n + k)$  time. In the worst case, when  $\varphi(t_q) = \Theta(n^2)$ , these query times are no better than brute force.

Kollios *et al.* [16] described a data structure for one-dimensional nearest-neighbor queries, but did not give any bound on the query time. Later, an efficient structure was given by Agarwal *et al.* [1] to answer  $\delta$ -approximate nearest-neighbor queries in  $O(n^{1/2+\varepsilon}/\sqrt{\delta})$  time using  $O(n/\sqrt{\delta})$  space. Their data structure can also be used to answer  $\delta$ -approximate farthest-neighbor queries; however, in both cases, the approximation parameter  $\delta$  must be fixed at preprocessing time. More practical data structures to compute exact and approximate  $k$ -nearest-neighbors were proposed by Procopiu *et al.* [20].

Basch *et al.* [9] described how to maintain the convex hull of a set of moving points in  $\mathbb{R}^2$  under the KDS framework. They showed that each kinetic event can be handled in logarithmic time, and the total number of kinetic events is  $O(n^{2+\varepsilon})$  if the trajectories are polynomials of fixed degree; the number of events is only  $O(n^2)$  if points move linearly [5]. With a kinetic convex hull at hand, one can easily answer a convex-hull query in  $O(\log n)$  time. However, as noted earlier, the kinetic convex hull can only answer queries on current configurations.

**Our results.** In this paper we present data structures for answering the five types of queries listed above. A main theme of our results is the existence of various tradeoffs: time-responsive data structures for which the query time depends on the value of  $t_q$ ; tradeoffs between query time and the accuracy of results; and tradeoffs between query time and the size of the data structure.

In Section 2, we describe a new time-responsive data structure for interval queries in  $\mathbb{R}^1$  and rectangle queries in  $\mathbb{R}^2$ . It uses  $O(n^{1+\varepsilon})$  space and can answer queries in  $O((\varphi(t_q)/n)^{1/2} + \text{polylog } n + k)$  time, where  $\varphi(t_q)$  is the number of events between  $t_q$  and the current time. To appreciate how this new query time improves over those of previously known data structures, let us examine two extreme cases. For near-future or recent-past queries, where  $\varphi(t_q) = O(n)$ , previous structures answer interval and rectangle queries in  $O(\log n)$  and  $O(\sqrt{n} \log n)$  time respectively, while our structures answer both queries in  $O(\text{polylog } n)$  time. In the worst case, where  $\varphi(t_q) = \Omega(n^2)$ , previous structures answer queries in  $O(n)$  time, which is no better than brute force, while our structures provide a query time of roughly  $O(\sqrt{n})$ , which is the best known for any structure of near-linear size. Using this result, we also obtain the first time-responsive structures for approximate nearest- and farthest-neighbor queries.

In Section 3, we present a data structure for answering  $\delta$ -approximate farthest-neighbor queries, which provides a tradeoff between the query time and the accuracy of the result. In particular, for any fixed parameter  $m > 1$ , we build a data structure of size  $O(m)$  in  $O(n + m^{7/4})$  time so that a  $\delta$ -approximate farthest-neighbor query can be answered in  $O(1/\delta^{9/4+\varepsilon})$  time, given a parameter  $\delta \geq 1/m^{1/4}$  as a part of the query. Note that these bounds are all independent of the number of points in  $S$ .

In Section 4, we present data structures for answering convex-hull queries. We first describe data structures that produce a continuous tradeoff between space and query time, by interpolating between one data structure of linear size and another structure with logarithmic query time. We then present a data structure that provides a tradeoff between efficiency and accuracy for approximate convex-hull queries.

## 2 Time-Responsive Data Structures

### 2.1 One-Dimensional Interval Queries

**Preliminaries.** Let  $L$  be a set of  $n$  lines in  $\mathbb{R}^2$ , let  $\Delta$  be a triangle, and let  $r$  be a parameter between 1 and  $n$ . A  $(1/r)$ -cutting of  $L$  within  $\Delta$  is a triangulation  $\Xi$  of  $\Delta$  such that each triangle of  $\Xi$  intersects at most  $n/r$  lines in  $L$ . Chazelle [13] described an efficient hierarchical method to construct  $1/r$ -cuttings. In this approach, one chooses a sufficiently large constant  $\rho$  and sets  $h = \lceil \log_\rho r \rceil$ . One then constructs a sequence of cuttings  $\Delta = \Xi_0, \Xi_1, \dots, \Xi_h = \Xi$ , where  $\Xi_i$  is a  $(1/\rho^i)$ -cutting of  $L$ .  $\Xi_i$  is obtained from  $\Xi_{i-1}$  by computing, for each triangle  $\tau \in \Xi_{i-1}$ , a  $(1/\rho)$ -cutting  $\Xi_i^\tau$  of  $L_\tau$  within  $\tau$ , where  $L_\tau \subseteq L$  is the set of lines intersecting  $\tau$ . Chazelle [13] proved that  $|\Xi_i| \leq (c_1\rho)^i + c_2\mu\rho^{2i}/n^2$  for each  $i$ , where  $c_1, c_2$  are constants and  $\mu$  is the number of vertices of the arrangement of  $L$  inside  $\Delta$ . Hence  $|\Xi| = O(r^{1+\varepsilon} + \mu r^2/n^2)$  for any  $\varepsilon > 0$ , provided that  $\rho$  is sufficiently large. This hierarchy of cuttings can be represented as a *cutting tree*  $\mathcal{T}$ , where each node  $v$  is associated with a triangle  $\Delta_v$  and the subset  $L_v \subseteq L$  of lines that intersect  $\Delta_v$ . The final cutting  $\Xi$  is the set of triangles associated with the leaves of  $\mathcal{T}$ .  $\Xi$  and  $\mathcal{T}$  can be constructed in time  $O(nr^\varepsilon + \mu r/n)$ .



**High-level structure.** Let  $S = \{p_1, \dots, p_n\}$  be a set of  $n$  points in  $\mathbb{R}^1$ , each moving linearly. We regard time  $t$  as an additional dimension; each moving point  $p_i$  traces a linear trajectory in the  $ty$ -plane, which we denote  $\ell_i$ . Let  $L = \{\ell_1, \dots, \ell_n\}$  be the set of all  $n$  trajectories, and let  $\mathcal{A}(L)$  denote the arrangement of lines in  $L$ . Answering an interval query for interval  $[y_1, y_2]$  and time stamp  $t_q$  is equivalent to reporting all lines in  $L$  that intersect the vertical line segment  $\{t_q\} \times [y_1, y_2]$ . We refer to this query as a *stabbing query*.

Each vertex in the arrangement  $\mathcal{A}(L)$  corresponds to a change of  $y$ -ordering for a pair of points in  $S$ , or in other words, a *kinetic event*. As in [2], we divide the  $ty$ -plane into  $O(\log n)$  vertical slabs as follows. Without loss of generality, suppose  $t = 0$  is the current time. Consider the partial arrangement of  $\mathcal{A}(L)$  that lies in the halfplane  $t \geq 0$ ; the other side of the arrangement is handled symmetrically. For each  $1 \leq i \leq \lceil \log_2 n \rceil$ , let  $\tau_i$  denote the  $t$ -coordinate of the  $(2^i n)$ th vertex to the right of the line  $t = 0$ , and set  $\tau_0 = 0$ . Using an algorithm of Brönnimann and Chazelle [10], each  $\tau_i$  can be computed in  $O(n \log n)$  time. For each  $i$ , let  $\mathcal{W}_i$  denote the half-open vertical slab  $[\tau_{i-1}, \tau_i) \times \mathbb{R}$ . By construction, there are at most  $2^i n$  kinetic events inside  $\bigcup_{j \leq i} \mathcal{W}_j$ . For each slab  $\mathcal{W}_i$ , we construct a separate *window data structure*  $\mathbb{W}_i$  for answering stabbing queries within  $\mathcal{W}_i$ . Our overall one-dimensional data structure consists of these  $O(\log n)$  window data structures.

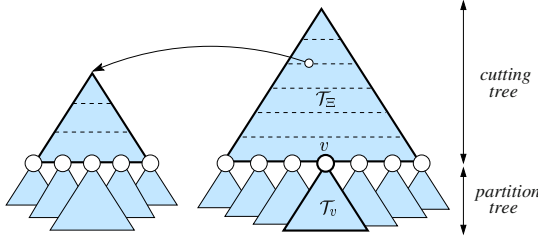
**Window data structure for one-sided queries.** Before we describe the window data structure  $\mathbb{W}_i$  for answering general stabbing queries in  $\mathcal{W}_i$ , we first describe a simpler structure  $\mathbb{W}_i^-$  to handle half-infinite query segments of the form  $\{t_q\} \times (-\infty, y]$ . Equivalently, we want to report the lines of  $L$  that lie below the query point  $(t_q, y)$ .

Set  $r = n/2^i$ . We construct a hierarchical  $(1/r)$ -cutting  $\mathcal{E}$  inside  $\mathcal{W}_i$ , as described at the beginning of this section. The cutting tree  $\mathcal{T}_{\mathcal{E}}$  of  $\mathcal{E}$  forms the top part of our data structure  $\mathbb{W}_i^-$ ; see Figure 1. Recall that any node  $u$  in  $\mathcal{T}_{\mathcal{E}}$  is associated with a triangle  $\Delta_u$  and the subset  $L_u \subseteq L$  of lines that intersect  $\Delta_u$ . For each leaf  $v$  of  $\mathcal{T}_{\mathcal{E}}$ , we build in time  $O(n^{1+\varepsilon})$  a partition tree [18] for the set of points dual to the lines in  $L_v$ . For a query point  $q = (t_q, y)$ , all  $k$  lines of  $L_v$  lying below  $q$  can be reported in time  $O((n/r)^{1/2} + k)$ . These partition trees form the bottom part of  $\mathbb{W}_i^-$ . Let  $p(u)$  denote the parent of a node  $u$  in  $\mathcal{T}_{\mathcal{E}}$ . At each node  $u$  of  $\mathcal{T}_{\mathcal{E}}$  except the root, we store an additional set  $J_u^- \subseteq L_{p(u)}$  of lines that cross  $\Delta_{p(u)}$  but lie below  $\Delta_u$ . If  $u$  is at level  $j$  in the cutting tree, then  $|J_u^-| \leq |L_{p(u)}| \leq n/\rho^{j-1}$ . This completes the construction of  $\mathbb{W}_i^-$ .

Let  $\mu \leq 2^i n$  denote the number of vertices of the arrangement  $\mathcal{A}(L)$  within the slab  $\mathcal{W}_i$ . The total space required by  $\mathcal{T}_{\mathcal{E}}$  is

$$O\left(\sum_{u \in \mathcal{T}_{\mathcal{E}}} |J_u^-|\right) = O\left(\sum_{j=1}^{\log_{\rho} r} ((c_1 \rho)^j + c_2 \mu \rho^{2j} / n^2) \cdot (n/\rho^{j-1})\right) = O(nr^{\varepsilon} + \mu r/n),$$

for any constant  $\varepsilon > 0$ , provided that the value of  $\rho$  is chosen sufficiently large. Each partition tree stored at a leaf of  $\mathcal{T}_{\mathcal{E}}$  requires  $O((n/r)^{1+\varepsilon})$  space, and there are  $O(r^{1+\varepsilon} + \mu r^2/n^2)$  such leaves. Substituting the values of  $\mu$  and  $r$ , we conclude



**Fig. 1.** One window structure in our one-dimensional time-responsive data structure.

that the total space required by  $\mathbb{W}_i^-$  is  $O(n^{1+\varepsilon})$ . Similar arguments imply that  $\mathbb{W}_i^-$  can be constructed in  $O(n^{1+\varepsilon})$  time.

Given a query point  $q = (t_q, y)$  that lies in slab  $W_i$ , we report the lines of  $L$  below  $q$  as follows. We search down the tree  $\mathbb{W}_i^-$  to locate the leaf node  $v$  of  $\mathcal{T}_\Xi$  whose associated triangle  $\Delta_v$  contains  $q$ . For each ancestor  $u$  of  $v$ , we report all lines in  $J_u^-$ . At the leaf node  $v$ , we use the partition tree of  $v$  to report all lines in  $L_v$  that lie below  $q$ . The correctness of the procedure is obvious. The total query time is  $O(\log_\rho r + (n/r)^{1/2} + k) = O(\log n + 2^{i/2} + k)$ , where  $k$  is the number of lines reported.

Our overall data structure consists of  $O(\log n)$  of these window structures. The total space and preprocessing time is  $O(n^{1+\varepsilon})$ ; the additional time required to locate the slab containing the query point is negligible. By construction, if the query point  $q$  lies in slab  $W_i$ , then  $2^{i-1}n \leq \varphi(t_q) < 2^i n$ .

**Lemma 1.** *Let  $S$  be a set of linearly moving points in  $\mathbb{R}$ . We can preprocess  $S$  in  $O(n^{1+\varepsilon})$  time into a data structure of size  $O(n^{1+\varepsilon})$  so that an interval query of the form  $(t_q; -\infty, y)$  can be answered in  $O(\log n + (\varphi(t_q)/n)^{1/2} + k)$  time, where  $k$  is the output size and  $\varphi(t_q)$  is the number of kinetic events between  $t_q$  and the current time.*

A symmetric structure can clearly be used to answer interval queries of the form  $(t_q; y, +\infty)$ .

**Window structure for query segments.** We now describe the window data structure  $\mathbb{W}_i$  for answering arbitrary interval queries. Our structure is based on the simple observation that a line  $\ell \in L$  intersects a segment  $\gamma = \{t_q\} \times [y_1, y_2]$  if and only if  $\ell$  intersects both  $\gamma^- = \{t_q\} \times (-\infty, y_2]$  and  $\gamma^+ = \{t_q\} \times [y_1, +\infty)$ . Our structure  $\mathbb{W}_i$  consists of two levels. The first level finds all lines of  $L$  that intersect the downward ray  $\gamma^-$  as a union of a few “canonical” subsets, and the second level reports the lines in each canonical subset that intersect the upward ray  $\gamma^+$ . See Figure 1.

We proceed as follows. Within each window  $W_i$ , we construct a cutting tree  $\mathcal{T}_\Xi$  on the trajectories  $L$  as before. For each node  $u \in \mathcal{T}_\Xi$ , let  $J_u^- \subseteq L$  be defined as above. We construct a secondary structure  $\mathbb{W}_i^+$  for  $J_u^-$  so that all  $k$  lines of  $J_u^-$  that lie above any query point can be reported in time  $O(\log n + 2^{i/2} + k)$ . Finally, for each leaf  $v$  of  $\mathcal{T}_\Xi$ , we construct in  $O(2^i \log n)$  time a partition tree on

the points dual to  $L_v$ , so that all  $k$  lines of  $L_v$  intersected by a vertical segment can be reported in time  $O(2^{i/2} + k)$ .

For a query segment  $\gamma = \{t_q\} \times [y_1, y_2]$ , we find the slab  $\mathcal{W}_i$  that contains  $\gamma$  and search  $\mathcal{T}_\Xi$  of  $\mathbb{W}_i$  with the endpoint  $(t_q, y_2)$ . Let  $v$  be the leaf of  $\mathcal{T}_\Xi$  whose triangle  $\Delta_v$  contains  $(t_q, y_2)$ . For each ancestor  $u$  of  $v$ , all the lines in  $J_u^-$  intersect the ray  $\gamma^-$ , so we query the secondary structure at  $u$  to report the lines in  $J_u^-$  that also intersect the ray  $\gamma^+$ . The total query time is  $O(\log^2 n + 2^{i/2} \log n + k)$ . By being more careful, we can improve the query time to  $O(\log^2 n + 2^{i/2} + k)$  without increasing the preprocessing time. If the query segment  $\gamma$  lies in slab  $\mathcal{W}_i$ , then  $2^{i-1}n \leq \varphi(t_q) < 2^i n$ . Thus, the query time can be rewritten as  $O(\log^2 n + (\varphi(t_q)/n)^{1/2} + k)$ .

Our analysis assumes that there are  $\Theta(2^i n)$  events between the current time and the right boundary of each slab  $\mathcal{W}_i$ . Thus, as time passes, we must occasionally update our data structure to maintain the same query times. Rather than making our structures kinetic, as in [1], we simply rebuild the entire structure after every  $\Theta(n)$  kinetic events. This approach immediately gives us an amortized update time of  $O(n^\varepsilon)$  per kinetic event; the update time can be made worst-case using standard lazy rebuilding techniques.

Putting everything together and choosing the value of  $\varepsilon$  carefully, we obtain the following theorem.

**Theorem 1.** *Let  $S$  be a set of  $n$  linearly moving points in  $\mathbb{R}$ . We can preprocess  $S$  in time  $O(n^{1+\varepsilon})$  into a data structure of size  $O(n^{1+\varepsilon})$ , such that an interval query on  $S$  can be answered in  $O(\log^2 n + (\varphi(t_q)/n)^{1/2} + k)$  time, where  $k$  is the output size and  $\varphi(t_q)$  is the number of kinetic events between the query's time stamp  $t_q$  and the current time. Moreover, we can maintain this data structure in  $O(n^\varepsilon)$  time per kinetic event.*

## 2.2 Two-Dimensional Rectangle Queries

We now describe how to generalize our interval-query data structure to handle two-dimensional rectangle queries. Let  $S = \{p_1, \dots, p_n\}$  be a set of  $n$  points in the plane, each moving with a fixed velocity. We again regard time  $t$  as an additional dimension. Let  $L = \{\ell_1, \ell_2, \dots, \ell_n\}$  be the set of linear trajectories of points of  $S$  in  $txy$ -space. Given a time stamp  $t_q$  and a rectangle  $R = R_x \times R_y$ , where  $R_x = [x_1, x_2]$  and  $R_y = [y_1, y_2]$ , a rectangle query is equivalent to reporting all lines in  $L$  that intersect the rectangle  $\{t_q\} \times R$  in  $txy$ -space. For any line  $\ell$ , let  $\ell^x$  and  $\ell^y$  denote the projections of  $\ell$  onto the  $tx$ -plane and  $ty$ -plane, respectively, and define  $L_x = \{\ell_1^x, \ell_2^x, \dots, \ell_n^x\}$  and  $L_y = \{\ell_1^y, \ell_2^y, \dots, \ell_n^y\}$ . Each vertex of  $\mathcal{A}(L_x)$  or  $\mathcal{A}(L_y)$  corresponds to a kinetic event. As for our one-dimensional structure, we divide  $txy$ -space into  $O(\log n)$  slabs along the  $t$ -axis such that the number of kinetic events inside the  $i$ th slab  $\mathcal{W}_i$  is  $O(2^i n)$ , and build a window data structure  $\mathbb{W}_i$  for each  $\mathcal{W}_i$ .

Observe that a line  $\ell$  intersects  $\{t_q\} \times R$  if and only if  $\ell^x$  intersects  $\{t_q\} \times R_x$  and  $\ell^y$  intersects  $\{t_q\} \times R_y$ . Thus we build a four-level data structure consisting of two one-dimensional structures described above, one on top of the other.

Specifically, we first build a two-level one-dimensional window data structure  $\mathbb{W}_i^x$  for  $L_x$  as described in Section 2.1, with one minor modification: all partition trees in the structure are replaced by two-level partition trees so that a rectangle query can be answered in time  $O(m^{1/2} + k)$ , where  $m$  is the number of lines on which the partition tree is built and  $k$  is the output size, as described in [1]. For each node  $u$  of the cutting tree in the secondary data structure of  $\mathbb{W}_i^x$ , let  $J_u^x$  be the canonical subset of lines stored at  $u$ . Set  $J_u^y = \{\ell_i^y \mid \ell_i^x \in J_u^x\}$ . We then build a one-dimensional window structure  $\mathbb{W}_{i,u}^y$  on  $J_u^y$  and store this structure at  $u$ .

A rectangle query can be answered by first finding the time interval  $\mathcal{W}_i$  containing the query rectangle  $\{t_q\} \times R$ . We then find the lines of  $L_x$  that intersect the segment  $R_x$ , as a union of canonical subsets. For each node  $u$  of  $\mathbb{W}_i^x$  in the output canonical subsets, we report all lines of  $J_u^y$  that intersect the segment  $R_y$  using the structure  $\mathbb{W}_{i,u}^y$ . Following standard analysis for multilevel data structures [3], we conclude the following.

**Theorem 2.** *Let  $S$  be a set of  $n$  linearly moving points in  $\mathbb{R}^2$ . We can preprocess  $S$  in time  $O(n^{1+\varepsilon})$  into a data structure of size  $O(n^{1+\varepsilon})$ , such that an rectangle query on  $S$  can be answered in  $O(\text{polylog } n + (\varphi(t_q)/n)^{1/2} + k)$  time, where  $k$  is the output size and  $\varphi(t_q)$  is the number of kinetic events between the query's time stamp  $t_q$  and the current time. Moreover, we can maintain this data structure in  $O(n^\varepsilon)$  time per kinetic event.*

Combining this result with techniques developed in [1], we can construct time-responsive data structures for  $\delta$ -approximate nearest- or farthest-neighbor queries, for any fixed  $\delta > 0$ . Specifically, we can prove the following theorem.

**Theorem 3.** *Let  $S$  be a set of  $n$  linearly moving points in  $\mathbb{R}^2$ . Given a parameter  $\delta > 0$ , we can preprocess  $S$  in time  $O(n^{1+\varepsilon}/\sqrt{\delta})$  into a data structure of size  $O(n^{1+\varepsilon}/\sqrt{\delta})$ , such that an approximate nearest- or farthest-neighbor query on  $S$  can be answered in  $O((\text{polylog } n + (\varphi(t_q)/n)^{1/2})/\sqrt{\delta})$  time, where  $\varphi(t_q)$  is the number of kinetic events between the query's time stamp  $t_q$  and the current time. Moreover, we can maintain this structure in  $O(n^\varepsilon/\sqrt{\delta})$  time per kinetic event.*

### 3 Approximate Farthest-Neighbor Queries

In this section, we describe a data structure for answering approximate farthest-neighbor queries that achieves a tradeoff between efficiency and accuracy. Unlike the approximate neighbor structure described in Theorem 3, the approximation parameter  $\delta$  can be specified at query time, and the cost of a query depends only on  $\delta$ , not on  $n$ . The general scheme is simple, but it crucially relies on the notion of  $\varepsilon$ -kernels introduced by Agarwal *et al.* [6].

For a unit vector  $u \in \mathbb{S}^{d-1}$ , the *directional width* of a point set  $P \subset \mathbb{R}^d$  along  $u$  is defined to be  $w_u(P) = \max_{p \in P} \langle u, p \rangle - \min_{p \in P} \langle u, p \rangle$ . A subset  $Q \subseteq P$  is called a  $\delta$ -kernel of  $P$  if  $w_u(Q) \geq (1 - \delta) \cdot w_u(P)$  for any unit vector  $u \in \mathbb{S}^{d-1}$ . A  $\delta$ -kernel of size  $O(1/\delta^{(d-1)/2})$  can be computed in  $O(n + 1/\delta^{d-1})$  time [12,24]. Using this general result and a result by Agarwal *et al.* on  $\delta$ -kernels for moving points [6, Theorem 6.2], we can prove the following lemma.

**Lemma 2.** *Let  $S = \{p_1, \dots, p_n\}$  be a set of  $n$  linearly moving points in  $\mathbb{R}^d$ . For any  $0 < \delta < 1$ , a subset  $Q \subseteq S$  of size  $O(1/\delta^{d+3/2})$  can be computed in  $O(n + 1/\delta^{2d+3})$  time, so that for any time  $t \in \mathbb{R}$  and any point  $q \in \mathbb{R}^d$ , we have*

$$(1 - \delta) \cdot \max_{p \in S} d(p(t), q) \leq \max_{p \in Q} d(p(t), q) \leq \max_{p \in S} d(p(t), q).$$

The subset  $Q$  in this lemma is referred to as a  $\delta$ -kernel for  $\delta$ -approximate farthest neighbors.

As we mentioned in Section 1, for a set of  $n$  linearly moving points in  $\mathbb{R}^2$ , one can build in  $O((n \log n)/\sqrt{\delta})$  time a data structure of size  $O(n/\sqrt{\delta})$  that can answer  $\delta$ -approximate farthest-neighbor queries in  $O(n^{1/2+\varepsilon}/\sqrt{\delta})$  time, for any fixed  $\delta$ . By combining this result with Lemma 2 (for  $d = 2$ ), we can construct in time  $O(n + 1/\delta^7)$  a data structure of size  $O(1/\delta^4)$  so that approximate farthest neighbors can be found in time  $O(1/\delta^{9/4+\varepsilon})$  for any  $\delta > 0$ .

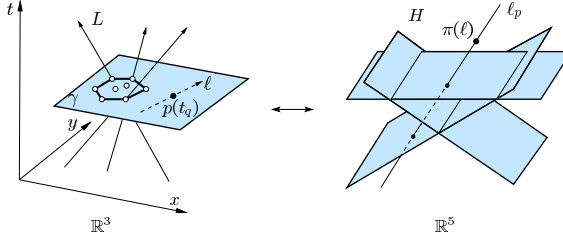
In order to allow  $\delta$  to be specified at query time, we build data structures for several different values of  $\delta$ . More precisely, fix a parameter  $m > 1$ . For each  $i$  between 0 and  $\lg m$ , let  $S_i$  be a  $\delta_i$ -kernel of  $S$ , where  $\delta_i = (2^i/m)^{1/4}$ . Lemma 2 immediately implies that all  $\lg m$  kernels  $S_i$  can be computed in  $O(n \log m + m^{7/4})$  time. We can reduce the computation time to  $O(n + m^{7/4})$  by constructing the kernels hierarchically: let  $S_0$  be a  $\delta_0$ -kernel  $S$ , and for each  $i > 0$ , let  $S_i$  be a  $O(\delta_i)$ -kernel of  $S_{i-1}$ . For each  $i$ , we build a data structure of size  $O(|S_i|/\sqrt{\delta_i}) = O(m/2^i)$  for finding  $\delta_i$ -approximate farthest neighbors in the kernel  $S_i$ . The total size of all  $O(\log m)$  data structures is  $O(m)$ . Given a  $\delta$ -approximate farthest-neighbor query, where  $(1/m)^{1/4} \leq \delta < 1$ , we first find an index  $i$  such that  $2\delta_i \leq \delta < 2\delta_{i+1}$ , and then query the data structure for the corresponding  $S_i$ . The returned point is a  $1 - (1 - \delta_i)^2 < 2\delta_i \leq \delta$  approximate farthest neighbor of the query point. The overall query time is  $O(|S_i|^{1/2+\varepsilon}/\sqrt{\delta_i}) = O((1/\delta)^{9/4+\varepsilon})$ .

**Theorem 4.** *Let  $S$  be a set of  $n$  linearly moving points in  $\mathbb{R}^2$ . For any parameter  $m > 1$ , a data structure of size  $O(m)$  can be built in  $O(n + m^{7/4})$  time so that for any  $(1/m)^{1/4} \leq \delta < 1$ , a  $\delta$ -approximate farthest-neighbor query on  $S$  can be answered in  $O((1/\delta)^{9/4+\varepsilon})$  time.*

## 4 Convex-Hull Queries

In this section, we describe data structures for convex-hull queries for linearly moving points in the plane that provide various tradeoffs. We begin with data structures that provide tradeoffs between space and query time by interpolating between two base structures—one with linear size, the other with logarithmic query time. We then show how to achieve a tradeoff between efficiency and accuracy, following the general spirit of Section 3.

**Trading off space for query time.** As in the previous section, let  $\ell_i$  be the line traced by  $p_i$  through  $xyt$ -space and let  $L = \{\ell_1, \ell_2, \dots, \ell_n\}$ . We orient each line in the positive  $t$ -direction. Given a time stamp  $t_q$  and a query point  $p \in \mathbb{R}^2$ , let  $\bar{p}$



**Fig. 2.** Plücker coordinates and convex-hull queries on moving points in  $\mathbb{R}^3$ .

denote the point  $(p, t_q) \in \mathbb{R}^3$  and let  $\gamma$  denote the plane  $t = t_q$  in  $\mathbb{R}^3$ . The convex-hull query now asks whether  $\bar{p} \in \text{conv}(\gamma \cap L)$ . Observe that  $\bar{p} \notin \text{conv}(\gamma \cap L)$  if and only if some line  $\ell \subset \gamma$  passes through  $\bar{p}$  and lies outside  $\text{conv}(\gamma \cap L)$ . Each of the two orientations of such a line  $\ell$  has the same relative orientation<sup>2</sup> with respect to every line in  $L$ ; see Figure 2.

We use Plücker coordinates to detect the existence of such a line. Plücker coordinates map an oriented line  $\ell$  in  $\mathbb{R}^3$  to either a point  $\pi(\ell)$  in  $\mathbb{R}^5$  (referred to as the *Plücker point* of  $\ell$ ) or a hyperplane  $\varpi(\ell)$  in  $\mathbb{R}^5$  (referred to as the *Plücker hyperplane* of  $\ell$ ). Furthermore, one oriented line  $\ell_1$  has positive orientation with respect to another oriented line  $\ell_2$  if and only if  $\pi(\ell_1)$  lies above  $\varpi(\ell_2)$  [14]. The following lemma is straightforward.

**Lemma 3.** *For any plane  $\gamma$  in  $\mathbb{R}$  and any point  $\bar{p} \in \gamma$ , the set of Plücker points of all oriented lines that lie in  $\gamma$  and pass through  $\bar{p}$  forms a line in  $\mathbb{R}^5$ .*

Let  $H = \{\varpi(\ell_i) \mid \ell_i \in L\}$  be the set of Plücker hyperplanes in  $\mathbb{R}^5$  of the oriented lines in  $L$ . For a query point  $\bar{p} = (t_q, p)$ , let  $\Lambda$  denote the line in  $\mathbb{R}^5$  of Plücker points of oriented lines that lie in the plane  $\gamma$  and pass through  $\bar{p}$ . A line  $\ell$  has the same relative orientation with respect to all lines in  $L$  if and only if  $\pi(\ell)$  lies either above every hyperplane in  $H$  or below every hyperplane in  $H$ . Thus, our goal is to determine whether there is any Plücker point on the line  $\Lambda$  that lies on the same side of every Plücker hyperplane in  $H$ . This can be formulated as a 5-dimensional linear-programming query, which can be solved directly using a data structure of Matoušek [19] to obtain the following result.<sup>3</sup>

**Theorem 5.** *Let  $S$  be a set of  $n$  linearly moving points in  $\mathbb{R}^2$ . Given a parameter  $n \leq s \leq n^2$ , a data structure of size  $O(s^{1+\epsilon})$  can be built in  $O(s^{1+\epsilon})$  time so that a convex-hull query on  $S$  can be answered in  $O((n/\sqrt{s}) \text{polylog } n)$  time.*

**Logarithmic query time.** If we are willing to use quadratic space, we can achieve  $O(\log n)$  query time without using the complicated linear-programming

<sup>2</sup> For any four points  $a, b, c, d \in \mathbb{R}^3$ , the *relative orientation* of the oriented line from  $a$  to  $b$  with respect to the oriented line from  $c$  to  $d$  is defined by the sign of the determinant of the 4-by-4 matrix  $\begin{pmatrix} a & b & c & d \\ 1 & 1 & 1 & 1 \end{pmatrix}$ .

<sup>3</sup> Data structures of Chan [11] and Ramos [21] can also be used for this purpose.

query structure. Basch *et al.* [9] described how to kinetically maintain  $\text{conv}(S(t))$  as  $t$  changes continuously, in  $O(\log^2 n)$  time per kinetic event. Let  $\mu$  denote the number of combinatorial changes to  $\text{conv}(S(t))$ ; since the points in  $S$  are moving linearly, we have  $\mu = O(n^2)$ . The kinetic convex hull may undergo some additional *internal* events, but the number of internal events is also  $O(n^2)$ . Thus, the entire kinetic simulation takes  $O(n^2 \log^2 n)$  time. Applying standard persistence techniques [23], we can record the combinatorial changes in  $\text{conv}(S(t))$  in  $O(n + \mu)$  space, so that for any time  $t_q$ , the combinatorial structure of  $\text{conv}(S(t_q))$  can be accessed in  $O(\log \mu) = O(\log n)$  time. With this combinatorial structure in hand, we can easily determine whether a point  $p$  lies inside or outside  $\text{conv}(S(t_q))$  in  $O(\log n)$  time.

**Theorem 6.** *Let  $S$  be a set of  $n$  linearly moving points in  $\mathbb{R}^2$ , and let  $\mu = O(n^2)$  be the total number of combinatorial changes to the convex hull of  $S$  over time. A data structure of size  $O(n + \mu)$  can be built in  $O(n^2 \log^2 n)$  time so that a convex-hull query on  $S$  can be answered in  $O(\log n)$  time.*

**Trading efficiency for accuracy.** Using ideas from the previous section, we can also build a data structure for *approximate* convex-hull queries. Recall that a subset  $Q \subseteq S$  is a  $\delta$ -kernel of a set  $S$  of moving points in  $\mathbb{R}^d$  if  $w_u(Q(t)) \geq (1 - \delta) \cdot w_u(S(t))$  for any time  $t$  and any unit vector  $u \in \mathbb{S}^{d-1}$ , where  $w_u(\cdot)$  is the directional width function defined in Section 3. If the points in  $S$  are moving linearly, then for any  $\delta$ , one can compute a  $\delta$ -kernel of  $S$  of size  $O(1/\delta^{3/2})$  in  $O(n + 1/\delta^3)$  time [6,12,24]. To establish a tradeoff between efficiency and accuracy for convex-hull queries, we proceed as in Section 3.

Let  $m > 1$  be a fixed parameter. For all  $i$  between 0 and  $\lg m$ , let  $\delta_i = (2^i/m)^{1/3}$ , and let  $S_i$  be a  $\delta_i$ -kernel of  $S$  of size  $O(1/\delta_i^{3/2}) = O((m/2^i)^{1/2})$ . We first compute all  $S_i$ 's in  $O(n + m)$  time as in Section 3, and then we build the logarithmic-query structure of Theorem 6 for each  $S_i$ . The size of one such data structure is  $O(|S_i|^2) = O(m/2^i)$ , so altogether these structures use  $O(m)$  space. To answer a  $\delta$ -approximate convex-hull query, where  $(1/m)^{1/3} \leq \delta < 1$ , we first find an index  $i$  such that  $\delta_i \leq \delta < \delta_{i+1}$ , and then query the corresponding data structure for  $S_i$ . The query time is  $O(\log |S_i|) = O(\log(1/\delta))$ .

**Theorem 7.** *Let  $S$  be a set of  $n$  linearly moving points in  $\mathbb{R}^2$ . For any parameter  $m > 1$ , a data structure of size  $O(m)$  can be built in  $O(n + m^{2+\epsilon})$  time so that for any  $(1/m)^{1/3} \leq \delta < 1$ , a  $\delta$ -approximate convex-hull query on  $S$  can be answered in  $O(\log(1/\delta))$  time.*

## References

1. P. K. Agarwal, L. Arge, and J. Erickson, Indexing moving points, *J. Comput. Syst. Sci.*, 66 (2003), 207–243.
2. P. K. Agarwal, L. Arge, and J. Vahrenhold, Time responsive external data structures for moving points, *Proc. 7th Workshop on Algorithms and Data Structures*, 2001, pp. 50–61.



3. P. K. Agarwal, J. Erickson, Geometric range searching and its relatives, in *Advances in Discrete and Computational Geometry* (B. Chazelle, J. Goodman, R. Pollack, eds.), American Mathematical Society, Providence, RI, 1999, 1–56.
4. P. K. Agarwal, J. Gao, and L. Guibas, Kinetic medians and *kd*-trees, *Proc. 10th European Symposium on Algorithms*, 2002, pp. 5–16.
5. P. K. Agarwal, L. Guibas, J. Hershberg, and E. Veach, Maintaining the extent of a moving point set, *Discrete Comput. Geom.*, 26 (2001), 253–274.
6. P. K. Agarwal, S. Har-Peled, and K. Varadarajan. Approximating extent measure of points. *Journal of the ACM*, to appear.
7. P. K. Agarwal and C. M. Procopiuc, Advances in indexing for moving objects, *IEEE Bulletin of Data Engineering*, 25 (2002), 25–34.
8. P. K. Agarwal and M. Sharir, Arrangements and their applications, in *Handbook of Computational Geometry* (J.-R. Sack and J. Urrutia, eds.), Elsevier Science Publishers, North-Holland, Amsterdam, 2000, 49–119.
9. J. Basch, L. Guibas, and J. Hersberger, Data structures for mobile data, *J. Algorithms*, 31(1) (1999), 1–28.
10. H. Brönnimann and B. Chazelle, Optimal slope selection via cuttings, *Comp. Geom. Theory & Appl.*, 10(1) (1998), 23–29.
11. T. M. Chan, Fixed-dimensional linear programming queries made easy, *Proc. 12th Annu. Sympos. Comput. Geom.*, 1996, pp. 284–290.
12. T. M. Chan, Faster core-set constructions and data stream algorithms in fixed dimensions, *Proc. 20th Annu. Sympos. Comput. Geom.*, 2004, pp. 152–159.
13. B. Chazelle, Cutting hyperplanes for divide-and-conquer, *Discrete Comput. Geom.*, 9 (1993), 145–158.
14. B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir, and J. Stolfi, Lines in space: Combinatorics and algorithms, *Algorithmica*, 15 (1996), 428–447.
15. A. Czumaj and C. Sohler, Soft kinetic data structures, *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, 2001, pp. 865–872.
16. G. Kollios, D. Gunopulos, and V. Tsotras, Nearest neighbor queries in a mobile environment, *Proc. Intl. Workshop on Spatiotemporal Database Management*, 1999, pp. 119–134.
17. G. Kollios, D. Gunopulos, and V. Tsotras, On indexing mobile objects, *Proc. ACM Sympos. Principles Database Syst.*, 1999, pp. 261–272.
18. J. Matoušek, Efficient partition trees, *Discrete Comput. Geom.*, 8 (1992), 315–334.
19. J. Matoušek, Linear optimization queries, *J. Algorithms*, 14 (1993), 432–448.
20. C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled, STAR-tree: an efficient self-adjusting index for moving points, *Proc 4th Workshop on Algorithms Engineering and Experiments*, 2002, pp. 178–193.
21. E. Ramos, Linear programming queries revisited, *Proc. 16th Annu. Sympos. Comput. Geom.*, 2000, pp. 176–181.
22. S. Saltenis, C. Jensen, S. Leutenegger, and M. López, Indexing the positions of continuously moving objects, *Proc. SIGMOD Intl. Conf. Management of Data*, 2000, pp. 331–342.
23. N. Sarnak and R. Tarjan, Planar point location using persistent search trees, *Communications of the ACM*, 29(7) (1986), 669–679.
24. H. Yu, P. K. Agarwal, R. Poredy, and K. R. Varadarajan, Practical methods for shape fitting and kinetic data structures using core sets, *Proc. 20th Annu. Sympos. Comput. Geom.*, 2004, pp. 263–272.



# Swap and Mismatch Edit Distance

Amihod Amir<sup>1\*</sup>, Estrella Eisenberg<sup>2\*\*</sup>, and Ely Porat<sup>2\*\*\*</sup>

<sup>1</sup> Department of Computer Science,  
Bar-Ilan University, 52900 Ramat-Gan, Israel  
and Georgia Tech  
Tel. (972-3)531-8770  
amir@cs.biu.ac.il

<sup>2</sup> Department of Computer Science,  
Bar-Ilan University, 52900 Ramat-Gan, Israel  
Tel. (972-3)531-7620  
estrigal@zahav.net.il, porately@cs.biu.ac.il

**Abstract.** There is no known algorithm that solves the general case of *approximate string matching problem* with the extended edit distance, where the edit operations are: insertion, deletion, mismatch, and swap, in time  $o(nm)$ , where  $n$  is the length of the text and  $m$  is the length of the pattern.

In the effort to study this problem, the edit operations were analysed independently. It turns out that the approximate matching problem with only the mismatch operation can be solved in time  $O(n\sqrt{m} \log m)$ . If the only edit operation allowed is the swap, then the problem can be solved in time  $O(n \log m \log \sigma)$ , where  $\sigma = \min(m, |\Sigma|)$ .

In this paper we show that the *approximate string matching problem* with the *swap* and *mismatch* as the edit operations, can be computed in time  $O(n\sqrt{m} \log m)$ .

## 1 Introduction

The last few decades have prompted the evolution of pattern matching from a combinatorial solution of the exact string matching problem to an area concerned with approximate matching of various relationships motivated by computational molecular biology, computer vision, and complex searches in digitized and distributed multimedia libraries [7]. To this end, two new paradigms were needed – “*Generalized matching*” and “*Approximate matching*”.

In generalized matching the input is still a text and pattern but the “matching” relation is defined differently. The output is all locations in the text where the pattern “matches” under the new definition of match. The different applications define the matching relation.

Even under the appropriate matching relation there is still a distinction between *exact matching* and *approximate matching*. In the latter case, a distance

---

\* Partly supported by NSF grant CCR-01-04494 and ISF grant 282/01.

\*\* This work is part of Estrella Eisenberg’s M.Sc. thesis.

\*\*\* Partially supported by GIF Young Scientists Program grant 2055-1168.6/2002.

function is defined on the text. A text location is considered a match if the distance between it and the pattern, under the given distance function, is within the tolerated bounds. Below are some examples that motivate approximate matching. In computational biology one may be interested in finding a “close” mutation, in communications one may want to adjust for transmission noise, in texts it may be desirable to allow common typing errors. In multimedia one may want to adjust for lossy compressions, occlusions, scaling, affine transformations or dimension loss.

The earliest and best known distance functions are Levenshtein’s *edit distance* [11] and the *Hamming distance*. Let  $n$  be the text length and  $m$  the pattern length. Lowrance and Wagner [12,13] proposed an  $O(nm)$  dynamic programming algorithm for the extended edit distance problem. In [10] the first  $O(kn)$  algorithm was given for the edit distance with only  $k$  allowed edit operations. Cole and Hariharan [5] presented an  $O(nk^4/m + n + m)$  algorithm for this problem. To this moment, however, there is no known algorithm that solves the general case of the extended edit distance problem in time  $o(nm)$ .

Since the upper bound for the edit distance seems very tough to break, attempts were made to consider the edit operations separately. If only mismatches are counted for the distance metric, we get the *Hamming distance*, which defines the *string matching with mismatches* problem. A great amount of work was done on finding efficient algorithms for string matching with mismatches [1,9,3]. The most efficient deterministic worst-case algorithm for finding the Hamming distance of the pattern at every text location runs in time  $O(n\sqrt{m \log m})$ . Isolating the swap edit operation yielded even better results [2,4], with a worst-case running time of  $O(n \log m \log \sigma)$ .

This paper integrates the above two results. Integration has proven tricky since various algorithms often involve different techniques. For example, there are efficient algorithms for string matching with don’t cares (e.g. [8]) and efficient algorithms for indexing exact matching (e.g. [14]), both are over 30 years old. Yet there is no known efficient algorithm for indexing with don’t cares. We provide an efficient algorithm for edit distance with *two* operations: *mismatch* and *swap*. Our algorithm runs in time  $O(n\sqrt{m \log m})$ . This matches the best known algorithm for Hamming distance matching. Indeed, a degenerate case of *swap and mismatch edit distance* matching, is the Hamming distance matching (for example, if all pattern symbols are different), so our algorithm is the best one can hope for considering the state-of-the-art. The techniques used by the algorithm are novel cases of overlap matching and convolutions.

## 2 Problem Definition

**Definition 1.** Let  $S = s_1 \dots s_n$  be a string over alphabet  $\Sigma$ . An edit operation  $E$  on  $S$  is a function  $E : \Sigma^n \rightarrow \Sigma^m$ . Let  $S = s_1 \dots s_n$  and  $T = t_1 \dots t_\ell$  be strings over alphabet  $\Sigma$ . Let  $OP$  be a set of local operations.  $OP$  is called the set of edit operations. The edit distance of  $S$  and  $T$  is the minimum number

$k$  such that there exist a sequence of  $k$  edit operations  $\langle E_1, \dots, E_k \rangle$  for which  $E_k(E_{k-1}(\dots E_1(T) \dots)) = S$ .

**Example:** The Lowrance and Wagner edit operations are:  $\{INS_{i,\sigma}, DEL_i, REP_{i,\sigma}, \text{ and } SWAP_i\}$ , where  
 $INS_{i,\sigma}(s_1 \dots s_n) = s_1 \dots s_i, \sigma, s_{i+1} \dots s_n$ ,  
 $DEL_i(s_1 \dots s_n) = s_1 \dots s_{i-1}, s_{i+1} \dots s_n$ ,  
 $REP_{i,\sigma}(s_1 \dots s_n) = s_1 \dots s_{i-1}, \sigma s_{i+1} \dots s_n$ , and  
 $SWAP_i(s_1 \dots s_n) = s_1 \dots s_{i-1}, s_{i+1}, s_i, s_i + 2 \dots s_n$ .

**Definition 2.** Let  $T = t_1 \dots t_n$  be a text string, and  $P = p_1 \dots p_m$  be a pattern string over alphabet  $\Sigma$ . The edit distance problem of  $P$  in  $T$  is that of computing, for each  $i = 1, \dots, n$ , the minimum edit distance of  $P$  and a prefix of  $t_i \dots t_n$ .

Lowrance and Wagner [12,13] give an  $O(nm)$  algorithm for computing the edit distance problem with the above four edit operations. To date, no better algorithm is known for the general case. We consider the following problem.

**Definition 3.** The swap mismatch edit distance problem is the following.

INPUT: Text string  $T = t_1 \dots t_n$  and pattern string  $P = p_1 \dots p_m$  over alphabet  $\Sigma$ .

OUTPUT: For each  $i = 1, \dots, n$ , compute the minimum edit distance of  $P$  and a prefix of  $t_i \dots t_n$ , where the edit operations are  $\{REP_{i,\sigma}, \text{ and } SWAP_i\}$ .

The following observation plays a role in our algorithm.

**Observation 1.** Every swap operation can be viewed as two replacement operations.

## 2.1 Convolutions

Convolutions are used for filtering in signal processing and other applications. A convolution uses two initial functions,  $t$  and  $p$ , to produce a third function  $t \otimes p$ . We formally define a discrete convolution.

**Definition 4.** Let  $T$  be a function whose domain is  $\{0, \dots, n-1\}$  and  $P$  a function whose domain is  $\{0, \dots, m-1\}$ . We may view  $T$  and  $P$  as arrays of numbers, whose lengths are  $n$  and  $m$ , respectively. The discrete convolution of  $T$  and  $P$  is the polynomial multiplication  $T \otimes P$ , where:

$$(T \otimes P)[j] = \sum_{i=0}^{m-1} T[j+i]P[i], \quad j = 0, \dots, n-m+1.$$

In the general case, the convolution can be computed by using the Fast Fourier Transform (FFT) [6] on  $T$  and  $P^R$ , the reverse of  $P$ . This can be done in time  $O(n \log m)$ , in a computational model with word size  $O(\log m)$ .

**Important Property:** The crucial property contributing to the usefulness of convolutions is the following. For every fixed location  $j_0$  in  $T$ , we are, in essence, overlaying  $P$  on  $T$ , starting at  $j_0$ , i.e.  $P[0]$  corresponds to  $T[j_0]$ ,  $P[1]$  to  $T[j_0 + 1]$ , ...,  $P[i]$  to  $T[j_0 + i]$ , ...,  $P[m - 1]$  to  $T[j_0 + m - 1]$ . We multiply each element of  $P$  by its corresponding element of  $T$  and add all  $m$  resulting products. This is the convolution's value at location  $j_0$ .

Clearly, computing the convolution's value for every text location  $j$ , can be done in time  $O(nm)$ . The fortunate property of convolutions over algebraically close fields is that they can be computed *for all*  $n$  text locations in time  $O(n \log m)$  using the FFT.

In the next few sections we will be using this property of convolutions to efficiently compute relations of patterns in texts. This will be done via *linear reductions* to convolutions. In the definition below  $\mathcal{N}$  represents the natural numbers and  $\mathcal{R}$  represents the real numbers.

**Definition 5.** Let  $P$  be a pattern of length  $m$  and  $T$  a text of length  $n$  over some alphabet  $\Sigma$ . Let  $R(S_1, S_2)$  be a relation on strings of length  $m$  over  $\Sigma$ . We say that the relation  $R$  holds between  $P$  and location  $j$  of  $T$  if  $R(P[0] \cdots P[m - 1], T[j]T[j + 1] \cdots T[j + m - 1])$ .

We say that  $R$  is linearly reduced to convolutions if there exist a natural number  $c$ , a constant time computable function  $f : \mathcal{N}^c \rightarrow \{0, 1\}$ , and linear time functions  $\ell_1^m, \dots, \ell_c^m$  and  $r_1^n, \dots, r_c^n$ ,  $\forall n, m \in \mathcal{N}$ , where  $\ell_i^m : \Sigma^m \rightarrow \mathcal{R}^m$ ,  $r_i^n : \Sigma^n \rightarrow \mathcal{R}^n$ ,  $i = 1, \dots, c$  such that  $R$  holds between  $P$  and location  $j$  in  $T$  iff

$$f(\ell_1^m(P) \otimes r_1^n(T)[j], \ell_2^m(P) \otimes r_2^n(T)[j], \dots, \ell_c^m(P) \otimes r_c^n(T)[j]) = 1.$$

Let  $R$  be a relation that is linearly reduced to convolutions. It follows immediately from the definition that, using the FFT to compute the  $c$  convolutions, it is possible to find all locations  $j$  in  $T$  where relation  $R$  holds in time  $O(n \log m)$ .

**Example:** Let  $\Sigma = \{a, b\}$  and  $R$  the equality relation. The locations where  $R$  holds between  $P$  and  $T$  are the locations  $j$  where  $T[j + i] = P[i]$ ,  $i = 0, \dots, m - 1$ . Fischer and Patterson [8] showed that it can be computed in time  $O(n \log m)$  by the following trivial reduction to convolutions.

Let  $\ell_1 = \chi_a$ ,  $\ell_2 = \chi_b$ ,  $r_1 = \chi_{\bar{a}}$ ,  $r_2 = \chi_{\bar{b}}$  where

$$\chi_\sigma(x) = \begin{cases} 1, & \text{if } x = \sigma; \\ 0, & \text{otherwise.} \end{cases} \quad \text{and} \quad \chi_{\bar{\sigma}}(x) = \begin{cases} 1, & \text{if } x \neq \sigma; \\ 0, & \text{otherwise.} \end{cases}$$

and where we extend the definition of the functions  $\chi_\sigma$  to a strings in the usual manner, i.e. for  $S = s_1 s_2 \dots s_n$ ,  $\chi_\sigma(S) = \chi_\sigma(s_1) \chi_\sigma(s_2) \dots \chi_\sigma(s_n)$ .

Let

$$f(x, y) = \begin{cases} 1, & \text{if } x = y = 0; \\ \text{otherwise.} \end{cases}$$

Then for every text location  $j$ ,  $f(\ell_1(P) \otimes r_1(T)[j], \ell_2(P) \otimes r_2(T)[j]) = 0$  iff there is an exact matching of  $P$  at location  $j$  of  $T$ .

### 3 Algorithm for Binary Alphabets

For simplicity's sake we solve the problem for binary alphabets  $\Sigma = \{0, 1\}$ . We later show how to handle larger alphabets.

When considering a binary alphabet, a swap operation can be effective only in the cases where the text has a pair 10 aligned with an 01 in the pattern, or vice versa. Therefore, we are interested in analysing the case of alternating sequences of zeros and ones. We define this concept formally, since it is key to the algorithm's idea.

An alternating segment of a string  $S \in \{0, 1\}^*$  is a substring alternating between 0s and 1s. A maximal alternating segment, or *segment* for short, is an alternating segment such that the character to the left of the leftmost character  $x$  in the alternating segment, if any, is identical to  $x$ , and similarly, the character to the right of the rightmost character  $y$ , if any, is identical to  $y$ .

Any string over  $\Sigma = \{0, 1\}$  can be represented as a concatenation of segments. We need to identify the cases where aligned text and pattern segments match via swap operations only and the cases where replacements are also necessary.

We now show the key property necessary to reduce swap matching to overlap matching. To this end we partition the text and pattern into segments.

**Example:**

Let  $P = 101000111010100110100111010101$

$P$ 's segments are: 1010 0 01 1 101010 01 1010 01 1 101010.

The following lemma was proven at [2].

**Lemma 1.** *The pattern does not (swap-) match in particular alignment if and only if there exists a segment  $A$  in the text and a segment  $B$  in the pattern such that: 1. The characters of  $A$  and  $B$  misalign in the overlap. 2. The overlap is of odd-length.*

The conditions of the above lemma are also useful for our problem.

**Lemma 2.** *The number of mismatches that are not part of a swap, is exactly the number of the overlaps that implement condition 1. and 2. of lemma 1.*

**Proof:** We will examine all possibilities:

1. Condition 1. of the lemma does not hold. Then there is no misalignment of the text. Indeed it matches the pattern.
2. Condition 1. holds but condition 2. does not. According to lemma 1 there is a swap-match.
3. If the two condition hold then either one of the two segments  $A$  and  $B$  is entirely contained in the other or the overlap is a real substring of both segments. For the first case we may assume, without loss of generality, that segment  $B$  of the pattern is contained in segment  $A$  of the text (the other case is treated in a similar fashion). The situation is that there is a misalignment and the overlap length is odd. Schematically, we have (with  $B$  and  $A$  boldfaced):

Pattern:  $\dots aabab \dots abaa \dots$   
 Text:  $\dots ababa \dots baba \dots$

Since swapping  $B$ 's edges will not help, the only swaps possible are internal to  $B$ . This means that there is exactly one element that remains mismatched after the swap.

The other situation is when the overlap is a real substring of both segments. We assume that  $B$  starts before  $A$  (the other case is handled in a similar fashion). The situation is:

Pattern:  $abab \dots abaa \dots$   
 Text:  $\dots bba \dots baba \dots$

Again it is clear that the only possible swaps are internal to  $B$ , leaving one element mismatched even after all possible swaps.  $\square$

**Corollary 1.** *The outline of our algorithm is as follows:*

1. For every text location  $i$ , count the number of mismatches  $m_i$  of the pattern starting at location  $i$  [1].
2. Partition the text and pattern into segments.
3. For every text location  $i$ , count the number  $r_i$  of odd-length misaligned overlaps. This is exactly the number of “real” (non-swap) mismatches.
4. The number of swap-errors at location  $i$  is  $s_i = (m_i - r_i)/2$ ,
5. The number of swap and mismatch edit errors at location  $i$  is  $r_i + s_i$ .

### 3.1 Grouping Text Segments by Parity of Starting and Ending Location

We follow some of the implementation ideas of [2]. However, there it was only necessary to check existence of odd-length mismatched overlaps, and we need to count them as well. The main idea we use is to separate the segments of the text and pattern into a small number of groups. In each of these groups it will be possible to count the required overlaps in time  $O(n\sqrt{m \log m})$  using a limited divide-and-conquer scheme based on polynomial multiplications (convolutions). In the subsections that follow we handle the different cases. Some of these cases necessitate new and creative uses of convolutions.

For checking the parity of the overlap, we need to know whether a text segment ends at an odd or even text location. Consequently, we define new texts where each text has *exactly* those segments of a given start and end parity, with all other text elements defined as  $\phi$  (*don't care*) which consequently never contribute an error. (In the polynomial multiplication there will always be a 0 in these text locations.) Henceforth, for short, we will talk of multiplying a text and pattern, by which we mean a polynomial multiplication, where each of the text and pattern is viewed as a vector of coefficients for the corresponding polynomials.

Consider the case  $T^{ti,tj}$  and  $P^{pi,pj}$  where  $ti = pi$ .

**Observation 2.** For every two segments,  $S_t$  in  $T^{ti,tj}$ , starting at location  $x$ , and  $S_p$  in  $P^{pi,pj}$ , starting at location  $y$ ,  $|x - y|$  is always even.

We are interested in the number of odd overlaps. We now show a convolution for which the resulting value at location  $i$  is  $n$  exactly if there is  $n$  odd-length overlaps with the pattern starting at location  $i$ . (The convolution with  $T^{eo}$  (or  $T^{oe}, P^{eo}, P^{oe}$ ) we need to do two convolutions, the first with  $T_1^{eo}$ , and the second with  $T_2^{eo}$  )

**The Convolution:** Pattern  $P' = p'_1 \cdots p'_m$  is constructed as follows:

$$p'_i = \begin{cases} 0, & \text{if } P^{pi,pj}[i] = \phi; \\ 1, & \text{otherwise.} \end{cases}$$

Text  $T' = t'_1 \cdots t'_n$  is constructed by replacing every  $\phi$  in  $T^{ti,tj}$  by 0, and every segment in  $T^{ti,tj}$  by a segment of alternating 1s and  $-1$ s, starting with 1. Then  $P'$  and  $T'$  are convolved.

**Lemma 3.** Let  $(T' \otimes P')[q]$  be the  $q$ th element in the result of the convolution.  $(T' \otimes P')[q]$  is equal to the number of odd overlaps of the relevant text segments and relevant pattern segments.

**Proof.** This follows from the definitions of convolutions, of  $T'$ , and of  $P'$  and from the observation that for all cases where the starting location of a pattern segment is smaller than the starting location of a text segment and the pair overlaps the contribution to the result of the convolution will be 1 if the length of the overlap is odd and 0 if it is even (since every text segment starts with a 1 and then alternates between  $-1$  and 1). Because of Observation 2, even when the text segment starts at a smaller location than the pattern segment, the difference between the starting locations has even length. Therefore in the area of the overlap, the text starts with a 1 and alternates between  $-1$  and 1. Thus the convolution gives us the desired result.  $\square$

Locations where  $(T' \otimes P')[q] = 0$  are locations without odd-overlap between relevant text and pattern segments.

This solves all eight cases of  $T^{ti,tj}$  and  $P^{pi,pj}$  where  $ti = pi$ . For the additional four cases where  $tj = pj$  we simply reverse the text and pattern to obtain the case considered above.

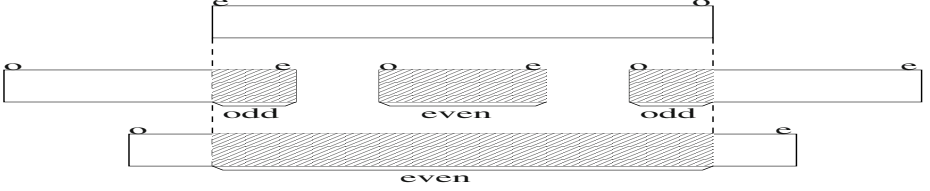
## 5 The Odd-Even Even-Odd Segments

Consider the case  $T_{1or2}^{oe}$  and  $P_{1or2}^{eo}$  (the case of  $T_{1or2}^{eo}$  and  $P_{1or2}^{oe}$  is symmetric).

**Terminology:** Let  $S_t$  be a text segment whose starting location is  $s_t$  and whose ending location is  $f_t$ . Let  $S_p$  be a pattern segment being compared to the text at starting position  $s_p$  and ending position  $f_p$ . If  $s_t < s_p < f_p < f_t$  then we say that  $S_t$  contains  $S_p$ . If  $s_p < s_t < f_t < f_p$  then we say that  $S_p$  contains  $S_t$ . If  $s_t < s_p < f_t < f_p$  then we say that  $S_t$  has a *left overlap* with  $S_p$ . If  $s_p < s_t < f_t < f_p$  then we say that  $S_t$  has a *right overlap* with  $S_p$ . We will sometimes refer to a left or right overlap as a *side overlap*.



**Observation 3.** For every two segments,  $S_t$  in  $T_{1or2}^{oe}$  and  $S_p$  in  $P_{1or2}^{eo}$  if either  $S_p$  is contained in  $S_t$  or  $S_t$  is contained in  $S_p$  then the overlap is of even length. If the overlap is a left overlap or right overlap then it is of odd length. All possible cases are shown in figure 1 below.



**Fig. 1.** The cases where the text segment starts at an odd location and ends at an even location, and the pattern segment does the opposite.

The correctness of the observation is immediate. Segments of these types have even length. Thus, if one contains the other the overlap is necessarily of even length. Conversely, in case of a left or right overlap, which we call side overlaps, the overlap starting and ending locations have the same parity, making the length of the overlap odd. Remember our desire is to count all locations where there are segments which have odd overlap. As before, we will use convolutions. The desired property for such a convolution is as follows.

### 5.1 The Convolutions for the Odd-Even Even-Odd Segments Case

**The Convolution:** Pattern  $T' = t'_1 \cdots t'_m$  is constructed as follows:

$$t'_i = \begin{cases} 0, & \text{if } T^{ti,tj}[i] = \phi; \\ 1, & \text{otherwise.} \end{cases}$$

Text  $P' = p'_1 \cdots p'_n$  is constructed by replacing every  $\phi$  in  $P^{pi,pj}$  by 0, and first and last place in every segment in  $P$  by 1, all the other places by 0.

For every two segments,  $S_t$  in  $T_{1or2}^{oe}$  and  $S_p$  in  $P_{1or2}^{eo}$  if  $S_p$  is contained in  $S_t$  it will add 2 to the convolution (even length overlap), if  $S_t$  is contained in  $S_p$  it will add 0 to the convolution. If the overlap is a left overlap or right overlap then it will add 1 to the convolution (each odd length overlap add 1 to the convolution).

Our convolution will add 1 for each odd-length overlap, and will add 2 for each segment of  $P$  that contains a segment of  $T$  ( $s_p < s_t < f_t < f_p$ ).

All we are left to do is count the number of cases of  $S_p$  contains  $S_t$ , in every text location  $i$ .

For each segment in the text and in the pattern we will add one place in the right (all the segments are larger in one). now all the segments in  $T$  and  $P$  (that were of even length) are odd length segments. All the segment in  $T$  are starting and ending in an odd index, all the segment in  $P$  are starting and ending in an even index.

**Observation 4.** *For every two segments,  $S_t$  in  $T_{1or2}^{oe}$  and  $S_p$  in  $P_{1or2}^{eo}$  if  $S_p$  was contained in  $S_t$  it is still contained in  $S_t$ . If  $S_t$  was contained in  $S_p$  then it is still contained in  $S_p$ . If the overlap was a left overlap (or right) overlap then it is still a left overlap (or right) overlap. In the first cases (contained or contain, the even overlaps) it becomes an odd overlap. In the second cases (right or left overlap, the odd overlaps) it becomes an even overlap.*

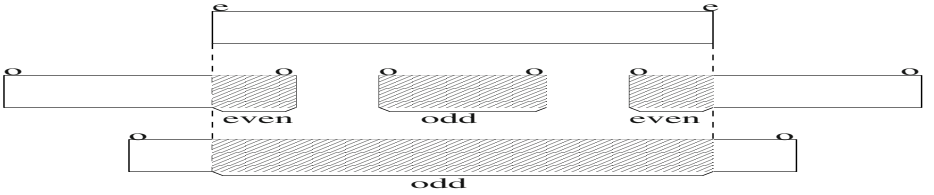
The correctness of the observation is immediate. We have four cases, the first case,  $S_p$  was contained in  $S_t$ ,  $S_t$  is larger by one, therefore,  $S_p$  is still contained in  $S_t$ . The second case  $S_t$  was contained in  $S_p$ ,  $S_t$  in the previous text ends in an even index, and  $S_p$ , in an odd index. Now both end in an odd index, therefore it is clear that if  $S_t$  was contained in  $S_p$ ,  $S_t$  is still contained in  $S_p$ . The third case, a right (left) overlap,  $S_t$  is larger by one from the right, therefore the overlap stay a right (left) overlap.

The new text and pattern are  $T^{oo}$  and  $P^{ee}$ , all we are left to do is count the number of odd length overlaps in the new text and pattern. This is done in the next section.

## 6 The Odd-Odd Even-Even Segments

Consider the case  $T^{oo}$  and  $P^{ee}$  (the case  $T^{ee}$  and  $P^{oo}$  is symmetric).

**Observation 5.** *For every two segments,  $S_t$  in  $T^{oo}$  and  $S_p$  in  $P^{ee}$  if either  $S_p$  is contained in  $S_t$  or  $S_t$  is contained in  $S_p$  then the overlap is of odd length. If the overlap is a left overlap or right overlap then it is of even length. All possible cases are shown in figure 2 below.*



**Fig. 2.** Every containment has odd length; every side overlap has even length.

The correctness of the observation is immediate. Segments of these types have odd lengths, thus if one contains the other the overlap is necessarily of odd length. Conversely, in case of a left or right overlap then the overlap starting and ending locations have opposite parity, making the length of the overlap even.

### 6.1 The Convolutions for the Odd-Odd Even-Even Segments Case

**The Convolution:** Pattern  $P' = p'_1 \cdots p'_m$  is constructed as follows:

$$p'_i = \begin{cases} 0, & \text{if } P^{pi,pj}[i] = \phi; \\ 1, & \text{otherwise.} \end{cases}$$

Text  $T' = t'_1 \cdots t'_n$  is constructed by replacing every  $\phi$  in  $T^{ti,tj}$  by 0, and every segment in  $T^{ti,tj}$  by a segment of alternating 1s and  $-1$ s, starting with 1. Then  $P'$  and  $T'$  are convolved.

For every two segments,  $S_t$  in  $T^{oo}$  and  $S_p$  in  $P^{ee}$  if  $S_p$  is contained in  $S_t$  it will add  $-1$  to the convolution (odd length overlap), if  $S_t$  is contained in  $S_p$  it will add 1 to the convolution. If the overlap is a left overlap or right overlap then it will add 0 to the convolution (each even length overlap add 0 to the convolution).

Our convolution will add 1 or  $-1$  for each odd-length overlap, and will add 0 for each segment  $S_t$  contains  $S_p$  or  $S_p$  contains  $S_t$ . Therefore, we will do two convolutions for each length  $l$  of segments in  $P$ , the first convolution will be with all the segments in  $T$  that their length is shorter than  $l$ , and the second convolution with the all the other segments in  $T$ .

The first convolution will give us  $-1$  for each odd length overlap (with the segments in  $T$  that are shorter than  $l$ ). The second convolution will give us 1 for each odd length overlap (with all the other segments in  $T$ ).

Now subtract the first convolution from the second convolution.

## 7 Solution for General Alphabet

Let  $\Sigma$  be a general alphabet and let  $\sigma = \min(n, |\Sigma|)$ .

Let  $K_i$  be the number of segments of length  $i$  in  $P$ . For each segment variety of length  $i$ , that appears in  $P$ , create a new pattern  $P'$ . The created pattern  $P'$  will contain all the original segments of length  $i$  in  $P$ , and all other symbols in the pattern, will be replaced by  $\phi$ . The number of lengths is  $\sqrt{m} (1 + 2 + \dots + \sqrt{m} = m)$ .

#### Solution for a specific length:

We divide the  $K_i$  segments of length  $i$  into two groups according to *frequency*. Every segment that contains more than  $\sqrt{K_i/\log m}$  appearances in pattern, is considered *frequent*.

#### Solution for non frequent segments of length $i$

Compare each (non frequent) segment with the all the relevant text segments (a relevant segment is segment that contains the same symbols as the pattern segment). This comparison is straightforward, since we know where each relevant text segment starts and ends. The time complexity is therefore  $O(1)$ , for each relevant segment.

**Total Time for a Nonfrequent Segment of length  $i$ :**  $\sigma R \sqrt{K_i/\log m} \leq n \sqrt{K_i/\log m}$ , where  $R$  is the number of relevant text segments.

**Solution for frequent segments of length  $i$ :** We reduce the problem into a  $\Sigma = \{0, 1\}$  problem. For each pair of symbols  $x_0, x_1$ , that are adjacent in the

pattern, we create a new pattern, replacing each appearance of  $x_0$  with 0 and each appearance of  $x_1$  with 1. All the other symbols are replaced by  $\phi$ .

The time complexity of a specific length  $i$ , in  $\Sigma = \{0, 1\}$  is  $n \log m$ . Since the number of appearances of that specific segment kind is at least  $\sqrt{K_i / \log m}$ , time complexity is  $n \log m \sqrt{K_i / \log m} = n \sqrt{K_i \log m}$ .

**Total Time:**

$$\begin{aligned} \text{Time} &\leq \sum_{i=2}^{\sqrt{m}} n \sqrt{K_i \log m} = n \sqrt{\log m} \sum_{i=2}^{\sqrt{m}} \sqrt{K_i} \leq n \sqrt{\log m} \sum_{i=2}^{\sqrt{m}} \sqrt{K_i} i^{\frac{1}{i}} \leq \\ &n \sqrt{\log m} \sqrt{\sum_{i=2}^{\sqrt{m}} K_i} i \sqrt{\sum_{i=2}^{\sqrt{m}} \frac{1}{i}} \leq n \sqrt{\log m} \sqrt{m} \sqrt{\log m} = n \sqrt{m} \log m. \quad \square \end{aligned}$$

## References

1. K. Abrahamson. Generalized string matching. *SIAM J. Comp.*, 16(6):1039-1051, 1987.
2. A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. Overlay matching. *Information and Computation*, 181(1):57-74, 2003.
3. A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with  $k$  mismatches. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 794-803, 2000.
4. A. Amir, M. Lewenstein, and E. Porat. Approximate swapped matching. *Information Processing Letters*, 83(1):33-39, 2002.
5. R. Cole and R. Hariharan. Approximate string matching: A faster simpler algorithm. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 463-472, 1998.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1992.
7. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.
8. M. J. Fischer and M. S. Paterson. String matching and other products. *Complexity of Computation, R. M. Karp (editor), SIAM-AMS Proceedings*, 7:133-125, 1974.
9. G. M. Landau and U. Vishkin. Efficient string matching with  $k$  mismatches. *Theoretical Computer Science*, 43:239-249, 1986.
10. G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157-169, 1989.
11. V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707-710, 1966.
12. R. Lowrance and R. A. Wagner. An extension of the string-to-string correction problem. *J. of the ACM*, pages 177-188, 1975.
13. R. A. Wagner. On the complexity of the extended string-to-string correction problem. In *Proc. 7th ACM STOC*, pages 218-223, 1975.
14. P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1-11, 1973.

# Path Decomposition Under a New Cost Measure with Applications to Optical Network Design

Elliot Anshelevich<sup>1</sup> and Lisa Zhang<sup>2</sup>

<sup>1</sup> Department of Computer Science, Cornell University, Ithaca NY.

Work partially done while visiting Bell Labs.

`eanshel@cs.cornell.edu`.

<sup>2</sup> Bell Laboratories, 600 Mountain Avenue, Murray Hill NJ.

`ylz@research.bell-labs.com`.

**Abstract.** We introduce a problem directly inspired by its application to DWDM (*dense wavelength division multiplexing*) network design. We are given a set of demands to be carried over a network. Our goal is to choose a route for each demand and to decompose the network into a collection of edge-disjoint simple paths. These paths are called *optical line systems*. The cost of routing one unit of demand is the number of line systems with which the demand route overlaps; our design objective is to minimize the total cost over all demands. This cost metric is motivated by the need to avoid O-E-O (*optical-electrical-optical*) conversions in optical transmission as well as to minimize the expense of the equipment necessary to carry the traffic.

For given line systems, it is easy to find the optimal demand routes. On the other hand, for given demand routes designing the optimal line systems can be NP-hard. We first present a 2-approximation for general network topologies. As optical networks often have low node degrees, we also offer an algorithm that finds the optimal solution for the special case in which the node degree is at most 3.

If neither demand routes nor line systems are fixed, the situation becomes much harder. Even for a restricted scenario on a 3-regular Hamiltonian network, no efficient algorithm can guarantee a constant approximation better than 2. For general topologies, we offer a simple algorithm with an  $O(\log K)$ - and an  $O(\log n)$ -approximation where  $K$  is the number of demands and  $n$  is the number of nodes. For rings, a common special topology, we offer a  $3/2$ -approximation.

## 1 Introduction

The constantly changing technology continues to present algorithmic challenges in optical network design. For example, the SONET (*Synchronous Optical Network*) technology has motivated a rich body of combinatorial work on rings, e.g. [4,10,19,20]; and the WDM (*wavelength division multiplexing*) technology has inspired numerous algorithms in coloring and wavelength assignment, e.g. [11, 12,15,17]. In this paper we introduce a novel problem directly motivated by its applications to DWDM (*dense WDM*) network design. At the essence of this

problem, however, lies a fundamental graph theory question. Our problem involves partitioning a graph into edge-disjoint paths with certain properties while minimizing a natural, yet so far largely unstudied, cost function.

The state-of-the-art DWDM technology allows more than one hundred wavelengths to be multiplexed on a single fiber strand and allows signals to travel thousands of kilometers optically. One design methodology in building an optical network using DWDM technology is to partition the network into a collection of paths which are called *optical line systems*. Such path decompositions have many advantages. For example, the linear structure simplifies wavelength assignment [21], engineering constraints [13], and requires less sophisticated hardware components. Therefore *line-based* optical networks are quite popular in today's optical industry.

In a line-based network an optical signal is transmitted within the optical domain as long as it follows a line system. However, when it switches from one line system to another, an O-E-O (*optical-electrical-optical*) conversion takes place. Such conversions are slow, expensive, and defeat the advantages of optical transmission. Therefore, our first design objective is to avoid O-E-O conversions as much as possible.

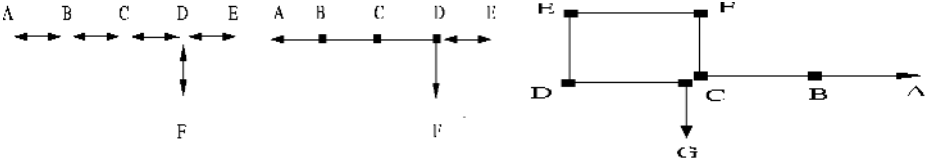
Naturally, our second objective is to design networks of low cost. As is usual in the case of optical network design, we cannot control the physical layout of the network since optical fibers are already installed underground. We are given a pre-existing network of *dark fiber*, i.e. optical fiber without any hardware for sending, receiving, or converting signals. The hardware cost consists of two parts, the *base cost* and the *transport cost*. The base cost refers to the cost of the equipment necessary to form the line systems. An optical line system begins with an *end terminal*, followed by an alternating sequence of fibers and OADM (*optical add/drop multiplexers*) and terminates with another end terminal. Roughly speaking, an OADM is built upon two back-to-back end terminals. Instead of terminating an optical signal an OADM allows the signal to pass through within the optical domain. An OADM costs about twice as much as an end terminal [7]. As a result, independent of how the network is partitioned into line systems, the base cost stays more or less the same. We can therefore ignore the base cost in our minimization. The transport cost refers to the cost of transmitting signals, in particular the cost incurred by the equipment needed to perform O-E-O conversions. Once the signal is formatted in the optical form it is transmitted along a line system for free. The number of O-E-O converters for a signal is linearly proportional to the number of line systems that the signal travels through. Therefore, minimizing O-E-O conversions serves the dual purpose of network cost minimization and providing (nearly) all-optical transmission.

*Our Model.* Let us describe the model in more detail. We are given a dark fiber network and a set of demands to be carried over the network. This network is modeled by an undirected graph  $G = (V, E)$ . Demands are undirected as well. Each demand is described by a source node and a destination node and carries one wavelength (or one unit) of traffic. We can have duplicate demands with the same source-destination pair. Let  $K$  be the sum of all demands. There are two

design components, *i*) partitioning the edge set  $E$  into a set of edge-disjoint line systems, and *ii*) routing each demand.

A routing path of a demand consists of one or more *transparent sections*, where each transparent section is the intersection of the routing path and a line system. It is worth pointing out here that if multiple units of demand switch from one line system to another, each demand unit requires its individual O-E-O converter. Therefore, our objective is minimizing the total number of transparent sections summed over all demands, which is equivalent to minimizing the network equipment cost and minimizing O-E-O conversions.

Figure 1 gives a dark fiber network and two of the many possible line system designs. Suppose one unit of demand travels from  $A$  to  $F$  and three units of demand travel from  $C$  to  $E$ . The only possible routing paths are  $ABCDF$  and  $CDE$  respectively. The solution on the left results in  $1 \times 4 + 3 \times 2 = 10$  transparent sections and the solution in the middle results in  $1 \times 1 + 3 \times 2 = 7$  transparent sections only.



**Fig. 1.** (Left) A dark fiber network. Each link is a separate line system. (Middle)  $ABCDF$  is a line system and  $DE$  is a line system. The arrows stand for end terminals and solid boxes stand for OADMs. (Right) An improper line system  $ABCFEDCG$ .

One complication in routing can arise under the following circumstances. When an optical signal travels between two nodes on a line system it stays within the optical domain only if it follows the line system path defined by the orientation of the OADMs. Otherwise, O-E-O conversions may take place. For example, the line system path in Figure 1(Right) is  $ABCFEDCG$ . To travel between  $B$  and  $G$  the direct path  $BCG$  is expensive since it does not follow the line system and incurs an O-E-O conversion at node  $C$ ; the “free” path  $BCFEDCG$  is non-simple since it visits  $C$  twice. Naturally, network carriers avoid expensive demand routes. On the other hand, they also need simple demand routes for reasons such as easy network management. We therefore restrict ourselves to line systems that induce simple demand routes only and call such line systems *proper*. More precisely, a *proper* line system corresponds to a path in which each node appears at most once, excluding its possible appearance(s) as the end points.

We also emphasize that wavelength assignment is not an issue in this context. A feasible wavelength assignment requires a demand to be on the same wavelength when it stays on one line system. However, it is free to switch to a different wavelength whenever it hops onto a different line system. Naturally,

for wavelength division multiplexing, two demands sharing a link need to have distinct wavelengths. Given such constraints, as long as the number of demands per link respects the fiber capacity, i.e. the number of wavelengths per fiber, a feasible wavelength can be found efficiently using the simple method for interval graph coloring. (See e.g. [6].) In this paper we assume infinite fiber capacity, since the capacity of a single fiber strand is as large as one terabit per second. From now on, we focus on our objective of cost minimization without having to consider wavelengths.

*Our Results.* In this paper we begin in Section 2 with optimizing line system design assuming simple demand routes are given. (We note that for given line systems finding the optimal routing paths is trivial.) For an arbitrary network with node degree higher than a certain constant  $c$ , finding the optimal line systems that minimize the total number of transparent sections is NP-hard. Fortunately, optical networks typically are sparse. Perhaps as our most interesting result, when the network has maximal node degree at most 3, we present a polynomial time algorithm that finds an optimal solution. For an arbitrary network topology, we also present a 2-approximation algorithm. This approximation is the best possible with the particular lower bound that we use.

In Section 3, we then focus on the general case in which neither routes nor line systems are given. We first show that this general case is much harder. In fact, even for a very restricted case on a 3-regular Hamiltonian network, no algorithm can guarantee a constant approximation better than 2. We then present a simple algorithm that guarantees an  $O(\log n)$ - and  $O(\log K)$ -approximation for arbitrary networks, where  $n$  is the number of nodes in the network and  $K$  is the number of demands.

In Section 3.2 we focus on the ring topology. Rings are of particular interest since often the underlying infrastructure is a ring for metro-area networks. We give a  $3/2$ -approximation. If the total demand terminating at each node is bounded by a constant, we present an optimal solution.

In the interest of space, we omit most of the proofs. The proofs, as well as a discussion on the relation of our problem to supereulerian graphs, can be found in the full version of this paper at [1].

*Related Work.* Until now, the design of line-based networks has only been studied empirically, e.g. [5,7]. However, the problem of ATM virtual path layout (VPL) is relevant. VPL aims to find a set of virtual paths in the network such that any demand route can be expressed as a concatenation of virtual paths, see e.g. [22,3]. The model for VPL differs from ours in several aspects. For example, our line systems form a *partition* of the network links whereas virtual paths are allowed to *overlap*. Also, our demand routes can enter and depart from any point along a line system whereas demand routes for VPL have to be concatenations of *entire* virtual paths. This makes a huge difference in the model, since the extra constraint that all routes must be concatenations of whole line systems can increase the optimal solution by a factor of  $K$ . Finally, we are concerned with minimizing the *total* number of transparent sections, while VPL papers are



traditionally concerned with minimizing the *maximum* hop count, with notable exceptions such as [9].

## 2 Designing Line Systems with Specified Routing

First we would like to point out that if line systems are given, it is easy to find the optimal route between any two nodes. The optimal route between any two nodes is the one that goes through as few line systems as possible and thus results in as few transparent sections as possible. We can find this route by calculating the shortest path according to an appropriate distance function.

If instead we are given the demand routes, and our goal is to find the optimal set of proper line systems, the problem becomes NP-hard. In fact, as we show in [1], even for networks with bounded degree the problem remains NP-hard. Therefore, we consider approximation algorithms.

### 2.1 2-Approximation for Proper Line Systems

We describe a *configuration* at each node, i.e. how the links incident to the node are connected to one another by line systems. For node  $u$  let  $E(u)$  be the set of links incident to  $u$ . Each link  $e \in E(u)$  is matched to at most one other link in  $E(u)$ . If  $e$  is matched to  $f \in E(u)$  it means  $e$  and  $f$  are on the same line system connected by an OADM at  $u$ . If  $e$  is not matched to any link in  $E(u)$  it means  $e$  terminates a line system at  $u$ . It is easy to see that the node configurations imply a set of paths that partition the network links. Of course, these resulting paths may not correspond to proper line systems.

Suppose every demand route is given. We offer a natural algorithm to configure each node. For every pair  $(u, v)$  and  $(v, w)$  of adjacent links, we define the *through traffic*  $T(uvw)$  as the number of demand units routed along  $u$ ,  $v$  and  $w$ . For each node  $v$ , we match the links incident to  $v$  such that the total through traffic along the matched link pairs is maximized. We refer to this algorithm as MAX THRU. Although MAX THRU may define closed loops or improper line systems, it has the following property.

**Lemma 1.** *For given demand routes, the total number of transparent sections generated by MAX THRU is a lower bound on the cost of the optimal feasible solution.*

To find proper line systems given simple demand routes, we begin with the MAX THRU algorithm and obtain a set of line systems that are not necessarily proper. If a line system is proper, we leave it as is. Otherwise, we cut the line system as follows. We traverse the line system from one end to the other and record every node that we visit in a *sequence*. If the line system is a closed loop we start from an arbitrary node and finish at the same node. (For example, the node sequence for the line system drawn in Figure 1(Right) is *ABCFEDCG*.) If a node  $u$  appears multiple times in the node sequence, we mark the first appearance of  $u$  with an open parenthesis “(”, the last appearance of  $u$  with

a closed parenthesis “)”, and every other appearance of  $u$  with a closed and an open parenthesis “(”. All these parentheses are labelled  $u$ . We *match* the  $i$ th open parenthesis labelled with  $u$  with the  $i$ th closed parenthesis also labelled with  $u$ . Each matching pair of parentheses represents a section of the line system that would induce non simple routing paths. We put down parentheses for every node that appears multiple times in the sequence.

We use these parentheses to determine where to cut an improper line system. We say two parentheses form an *inner-most pair* if the left parenthesis is “(”, the right one is “)” and they do not contain any other parenthesis in between. Note the two parentheses in an inner-most pair do not have to have the same label. We now find the pair of inner-most parentheses that is also left most, and cut the sequence at the node  $v$  where the closed parenthesis from the selected pair sits. We remove every matching pair of parentheses that contains  $v$ . We repeat the above process until no parentheses are left. We refer to this algorithm as CUT PAREN, which has 2 desirable properties as stated in Lemmas 2 and 3.

**Lemma 2.** *The CUT PAREN algorithm defines a set of proper line systems.*

**Lemma 3.** *Given simple routing paths, each transparent section defined by MAX THRU is cut into at most 2 pieces by CUT PAREN.*

By Lemma 2 CUT PAREN defines a set of proper line systems. By Lemmas 1 and 3 we know that CUT PAREN is at most twice as expensive as any optimal solution. Hence,

**Theorem 1.** *Given simple routing paths, CUT PAREN is a 2-approximation algorithm.*

Using the lower bound given by Lemma 1, no algorithm can beat the approximation ratio of 2 since the optimal solution can cost twice as much as the infeasible solution generated by MAX THRU. For example, this can happen on a cycle with a unit demand between any two neighboring nodes  $u$  and  $v$  and the demand being routed along the “long” path excluding link  $uv$ .

So far we have presented an algorithm that can cut an improper line system with a guaranteed approximation ratio of 2. In fact, we can always cut an improper line system  $L$  optimally as shown in [1]. Unfortunately, we do not know if this can lead to a better approximation ratio than 2. However, we have an example that shows that even if we cut the improper line systems generated by MAX THRU optimally, we can still end up with a solution that costs  $5/4$  as much as the optimal solution with proper line systems.

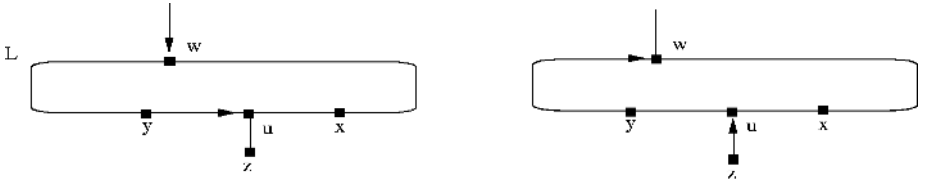
## 2.2 Networks of Node Degree 3

The situation is far better when the network only has nodes of low degree. In particular, if the network has maximal degree of at most 3, we can indeed find an optimal set of proper line systems for any given simple demand routes. This is fortunate since optical networks are typically sparse, and only have a small

number of nodes of degree greater than 3. Again, we begin with MAX THRU. Since the node degree is at most 3, we observe that a resulting line system is improper only if it is a simple closed loop, i.e. a closed loop that induces simple routes only. We now examine each node  $u$  on the loop, which we call  $L$ . Let  $x$  and  $y$  be  $u$ 's neighboring nodes on the loop  $L$  and  $z$  be  $u$ 's neighboring node off the loop (if such a  $z$  exists). Without loss of generality, let us assume the through traffic  $T(xuz)$  is no smaller than the through traffic  $T(yuz)$ . (If  $z$  does not exist, then  $T(xuz) = T(yuz) = 0$ .) An alternate configuration at  $u$  would be to cut  $xuy$  at  $u$  and connect  $xuz$ , and we refer to this operation as a *swap*. This swap would increase the total number of transparent sections by  $I(u)$  where  $I(u) = T(xuz) - T(xuy)$ . If node  $u' \in L$  has the minimum increase  $I(u')$  then we perform the swap operation at  $u'$ . We swap exactly one node on each closed loop. We refer to this algorithm as GREEDY SWAP. Since every node has degree at most 3, GREEDY SWAP eliminates all loops and never creates new ones. Therefore, the GREEDY SWAP algorithm defines a set of proper line systems.

**Theorem 2.** *The GREEDY SWAP algorithm is optimal for given demand routing on networks with node degree at most 3.*

*Proof.* We first note that no matter how GREEDY SWAP breaks ties in decision making the resulting solutions have the same total number of transparent sections. Given a solution by an optimal algorithm OPT, we now show that GREEDY SWAP produces an identical solution under some tie breaking. In particular, when executing GREEDY SWAP let us assume that whenever the algorithm is presented with a tie-breaking choice, it always arranges a node to be configured in the same way as in OPT, if possible.



**Fig. 2.** (Left) OPT. (Right) Result of GREEDY SWAP.

Now let us compare the line systems produced by GREEDY SWAP against the optimal solution. We first examine each node  $u$  for which MAX THRU and GREEDY SWAP have the same configuration. We claim that OPT has the same configuration at  $u$  as well. Let us focus on the case in which  $u$  has three neighbors  $x$ ,  $y$  and  $z$ . (The cases in which  $u$  is degree 1 or 2 are simpler.) If MAX THRU does not connect  $xuy$ ,  $xuz$  or  $yuz$ , OPT must have the same configuration at  $u$  since the through traffic  $T(xuy)$ ,  $T(xuz)$  and  $T(yuz)$  must be all zero and OPT is used for tie breaking. Otherwise, let us assume without loss of generality that MAX THRU connects  $xuy$ . For the purpose of contradiction, let us assume that OPT

connects  $xuz$ . By the construction of MAX THRU and the tie breaking rule, we have  $T(xuy) > T(xuz)$ . If OPT reconfigures node  $u$  by connecting  $xuy$  instead, a loop  $L$  must be formed or else we would have a better solution than OPT (see Figure 2). On the loop  $L$  there must be a node  $v \neq u$  such that GREEDY SWAP and OPT configure  $v$  differently, since otherwise GREEDY SWAP would contain this loop  $L$ . Let  $w$  be the first such node, i.e. all nodes on  $L$  between  $u$  and  $w$  in the clockwise direction are configured the same by GREEDY SWAP and OPT.

If  $w$  has the same configuration in MAX THRU and GREEDY SWAP, then by the definition of MAX THRU and by the tie breaking rule, the configuration of  $w$  with MAX THRU must be strictly better than its configuration with OPT. Hence, by reconfiguring OPT at nodes  $u$  and  $w$  like GREEDY SWAP, we get a better solution than OPT which is a contradiction. Therefore, as shown in Figure 2,  $w$  must be a node that was cut by GREEDY SWAP on a loop generated by MAX THRU that included  $u$ , which implies that  $I(w) \leq I(u)$ . In fact, it must be the case that  $I(w) < I(u)$  by the tie breaking rule. Therefore, if we reconfigure OPT at nodes  $u$  and  $w$  like GREEDY SWAP, we once again get a better solution than OPT. This is a valid solution, since we cannot create any loops in OPT by reconfiguring both  $u$  and  $w$ . Hence, if MAX THRU and GREEDY SWAP have the same configuration for node  $u$ , OPT has the same configuration at  $u$  as well.

We finally examine each node  $u$  that MAX THRU and GREEDY SWAP configure differently. This node  $u$  can only exist on a closed loop  $L$  created by MAX THRU and each closed loop can only have one such node. For every node  $v \in L$  and  $v \neq u$ , MAX THRU and GREEDY SWAP configure  $v$  in the same way by the construction of GREEDY SWAP. Hence, by our argument above, OPT configures  $v$  in the same way as well. Since OPT contains no loops, it has to configure node  $u$  like GREEDY SWAP.  $\square$

### 3 The General Case

The general case in which neither demand routes nor line systems are given is much harder. Contrary to Theorem 2, the general case is NP-hard even for restricted instances on 3-regular networks. Furthermore, we also have a stronger result as stated in Theorem 3. The proof of this theorem uses a result by Bazgan, Santha, and Tuza[2], which states that the longest path problem cannot be approximated within a constant factor in 3-regular Hamiltonian graphs.

**Theorem 3.** *If routes are not specified, no algorithm can guarantee a constant approximation ratio better than 2 for our problem even if the network is 3-regular and Hamiltonian and even if all demands originate from the same source node.*

#### 3.1 A Logarithmic Approximation

In this section we present a logarithmic approximation for the general case. Let  $T$  be any tree that spans all source nodes and destination nodes. We choose an arbitrary node  $r$  in  $T$  to be the root of the tree. We route every demand from its source to  $r$  and then to its destination along edges of  $T$ .

We arrange the line systems as follows. Consider a node  $y$ , and let  $z$  be its parent node. If  $x$  is the child node of  $y$  that has the largest subtree then the only line system through node  $y$  is along  $zyx$ . The number of transparent sections from any node to the root  $r$  is at most  $\log n$  where  $n$  is the number of nodes in  $T$ , since whenever the number of transparent sections increases by 1, the size of the subtree is cut by at least a half (also true because the path decomposition we have formed is a *caterpillar decomposition* [14]). Therefore, for any demand to go from its source node to its destination node the number of transparent sections is at most  $2 \log n$ .

If we consider the “size” of a subtree as the number of destination nodes in the subtree rooted at  $x$ , then the same construction of the line systems as above would bound the maximum number of transparent sections to  $2 \log(2K)$  where  $K$  is the total number of demands. Since the total cost of any solution must be at least  $K$ , we have the following theorem.

**Theorem 4.** *For the general case with unspecified demand routes, we can find a  $2 \log n$ -approximation and a  $2 \log(2K)$  approximation in polynomial time.*

Notice that this approximation ratio is tight comparing against the lower bound,  $K$ , that we are using, as evidenced by a balanced binary tree with one unit demand between the root and every leaf node.

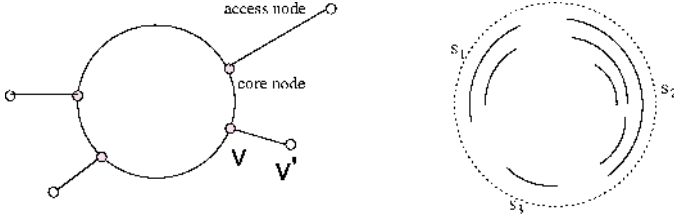
### 3.2 Rings

The ring topology is of particular interest since the underlying infrastructure of metro-area networks is often a ring [18]. In this context, we have a set of *core* nodes connected in a ring and each core node is attached to an *access* node, as in Figure 3(Left). Among other things, the access nodes have the capability of multiplexing low-rate local traffic streams into high-rate optical signals. Such access nodes are particularly useful when end users are far away from the core ring. In this ring network, all demands are between access nodes. It is easy to see that each demand has two routes, each access node has one configuration and each core node has three configurations which completely determine the layout of the line systems. Under this constrained scenario, it is reasonable to expect that we can do much better than the logarithmic approximation of Section 3.1. Indeed, we obtain the following result.

**Theorem 5.** *In ring networks, with no multiple demands between the same node pair, we can find a  $3/2$ -approximation in polynomial time.*

The proof of this theorem is somewhat involved and can be found in the full version of this paper at [1]. Here we will give the general idea of it.

To prove the above theorem we show that either we can find an optimal solution efficiently or the *whole ring solution* serves as a  $3/2$  approximation. In the whole ring solution, all the core nodes form a single line system around the ring and it is cut open at an arbitrary core node. It is easy to see that in the



**Fig. 3.** (Left) A ring network. (Right) A consistent routing of  $S$  and the sets  $S_j$ .

whole ring solution we can route the demands so that each demand requires 3 transparent sections.

Before we describe our algorithm we begin by describing *crossing*, *non-crossing* demands and their properties. For a demand that starts at  $v'$  and ends at  $w'$ , we refer to  $v$  and  $w$  (the corresponding core nodes) as the *terminals* of this demand. We call a pair of demands a *crossing pair* if the four terminal nodes are distinct and one terminal of a demand always appears in between the two terminals of the other demand. Now we consider a pair of non-crossing demands. If their routes do not overlap or overlap at one contiguous section only, then we say the demand paths are *consistent*. If the demand paths of all pairs of non-crossing demands are consistent, we say the solution has consistent routing. We establish the following two properties about routing non-crossing demands consistently.

**Lemma 4.** *There is an optimal solution that we call  $OPT$ , for which the routing is consistent.*

**Lemma 5.** *For a set of mutually non-crossing demands  $S$ , the number of ways to route the demands consistently is linear in  $|S|$ .*

*The Algorithm.* We now describe our  $3/2$ -approximation algorithm. From the set of demands, we take out crossing pairs of demands, one pair at a time in arbitrary order, until we have no crossing pairs left. Let  $C$  be the set of crossing pairs of demands that we have removed, and  $S$  be the set of non-crossing demands that are left. By Lemma 4 we know that  $OPT$  must use consistent routing, and by Lemma 5 we can efficiently enumerate all possible consistent routings for demands in  $S$ . Hence, if  $C$  is empty we know how to find an optimal solution and we are done. Otherwise, we assume from now on that we know how  $OPT$  consistently routes the demands in  $S$ . We also know that the routing of  $S$  results in a collection of disjoint intervals  $I_1, I_2, \dots$  around the ring, where each interval  $I_j$  consists of links that are on the routes of demands of  $S$ . Let  $S_j \subseteq S$  be the set of demands whose routes are included in  $I_j$ , as illustrated in Figure 3(Right). We denote by  $c(S_j)$  the total number of transparent sections that the demands in the set  $S_j$  use in  $OPT$ . Each set  $S_j$  obeys the following lemma.

**Lemma 6.** *We have  $c(S_j) \geq 2|S_j| - 1$ . The equality happens only if at each border node  $s$  of  $I_j$ , the configuration of  $s$  is  $s'sv$  where  $v \in I_j$ . Finally, if  $I_j$  is the entire ring, then  $c(S_j) \geq 2|S_j|$ .*

If  $c(S_j) = 2|S_j| - 1$  and the interval  $I_j$  causes no extra jumps for any demand in  $C$ , we call the set  $S_j$  *thrifty*. We can prove the following key lemma.

**Lemma 7.** *If  $OPT$  does not contain a thrifty set, the whole ring solution is a  $3/2$ -approximation. If  $OPT$  contains a thrifty set, then we can find the routing of all demands of  $C$  in  $OPT$ .*

Using this lemma, we consider each set  $S_j$ , and find a routing of all demands assuming it is thrifty. Given the routes of all demands, we can find the optimal line systems by Theorem 2. If none of the  $S_j$  are thrifty, we use the whole ring solution which has a cost of  $3K$ . Now that we have obtained a total of  $O(x)$  solutions where  $x$  is the number of sets  $S_j$ , we choose the least expensive one among them. If it is the case that  $OPT$  contains no thrifty set, the solution we have found is no worse than the whole ring solution which guarantees a  $3/2$ -approximation by Lemma 7. On the other hand, if  $OPT$  contains some thrifty set, our solution is guaranteed to find this set through enumeration and is therefore optimal. This proves Theorem 5.

This finishes our  $3/2$ -approximation algorithm for rings. However, if the demand originating at every node of the ring is bounded by some constant  $k$ , we can find the optimal solution in polynomial time, even if the same source-destination pair can have multiple demands.

## 4 Variations and Open Problems

In this paper, we introduced a new cost metric which measures the cost of routing a demand by the number of line systems that the demand travels through. We presented a collection of results. However, many questions remain open and some variations deserve further attention.

The algorithm we have presented for line system partitioning with given demand routes guarantees a 2-approximation. However, we have no example in which our algorithm gives an approximation ratio worse than  $5/4$ , or any reason to believe that a different algorithm may not do even better. If the demand routes are not given, we only have an  $O(\log n)$ -approximation that compares against a rather trivial lower bound,  $K$ , on  $OPT$ . In fact, we know that on a binary tree  $OPT$  can be as much as  $\Theta(K \log n)$ . Therefore, a much better approximation seems likely. The only inapproximability result says that the approximation ratio cannot be better than 2, so even a small constant approximation may be possible.

It is worth noting that our model makes several simplifications. Most notably, we assume that optical fibers have infinite capacity and that line systems can have arbitrary length. It would be quite interesting to introduce one or both of these complications into our model and see if good algorithms are still possible.

**Acknowledgements.** We would like to thank Chandra Chekuri, Jon Kleinberg, and Peter Winkler for productive and illuminating discussions, and thank Steve Fortune for his insight in defining the model.

## References

1. E. Anshelevich, L. Zhang. Path Decomposition under a New Cost Measure with Applications to Optical Network Design. (full version)  
<http://www.cs.cornell.edu/people/eanshel/>
2. C. Bazgan, M. Santha, and Z. Tuza. On the approximability of finding a(nother) Hamiltonian cycle in cubic Hamiltonian graphs. *Journal of Algorithms*, 31:249 – 268.
3. J.-C. Bermond, N. Marlin, D. Peleg, S. Pérennes. Virtual path layouts with low congestion or low diameter in ATM networks. In *Proceedings of 1ère Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, 1999.
4. S. Cosares and I. Saniee. An optimization problem related to balancing loads on SONET rings. *Telecommunications Systems*, 3:165 – 181, 1994.
5. B. Doshi, R. Nagarajan, N. Blackwood, S. Jothipragsam, N. Raman, M. Sharma, and S. Prasanna. LIPI: A lightpath intelligent instantiation tool: capabilities and impact. *Bell Labs Technical Journal*, 2002.
6. P. Fishburn. *Interval orders and interval graphs*. Wiley and Sons, New York, 1985.
7. S. Fortune, W. Sweldens, and L. Zhang. Line system design for DWDM networks. Submitted.
8. M. R. Garey and D. S. Johnson. *Computers and intractability - A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.
9. O. Gerstel and A. Segall. Dynamic maintenance of the virtual path layout. In *Proceedings of IEEE INFOCOM'95*, April 1995.
10. S. Khanna. A polynomial-time approximation scheme for the SONET ring loading problem. *Bell Labs Technical Journal*, 1997.
11. J. Kleinberg and A. Kumar. Wavelength conversion in optical networks. In *SODA 1999*, pages 566 – 575.
12. V. Kumar and E. Schwabe. Improved access to optical bandwidth in trees. In *SODA 1997*, pages 437 – 444.
13. W. Lee. Personal communication. 2003.
14. J. Matoušek. On embedding trees into uniformly convex banach spaces. *Israel Journal of Mathematics*, 114:221 – 237, 1999.
15. M. Mihail, C. Kaklamanis, and S. Rao. Efficient access to optical bandwidth. In *FOCS 1995*, pages 548 – 557.
16. C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Dover Publications, Mineola, New York, 1998.
17. P. Raghavan and E. Upfal. Efficient routing in all-optical networks. In *STOC 1994*, pages 134–143.
18. R. Ramaswami and K. Sivarajan. *Optical networks A practical perspective*. Morgan Kaufmann, San Francisco, CA, 1998.
19. A. Schrijver, P. D. Seymour, and P. Winkler. The ring loading problem. *SIAM Journal of Discrete Math*, 11(1):1 – 14, 1998.
20. G. Wilfong and P. Winkler. Ring routing and wavelength translation. In *SODA 1998*, pages 333 – 341.
21. P. Winkler and L. Zhang. Wavelength assignment and generalized interval graph coloring. In *SODA 2003*.
22. S. Zaks. Path Layout in ATM Networks - A Survey. In *Networks in Distributed Computing*, M. Mavronicolas, M. Merritt, and N. Shavit, Eds., DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, 1998, pp. 145-160.



# Optimal External Memory Planar Point Enclosure

Lars Arge<sup>1\*</sup>, Vasilis Samoladas<sup>2</sup>, and Ke Yi<sup>1\*</sup>

<sup>1</sup> Department of Computer Science, Duke University, Durham, NC 27708, USA.  
`{large,yike}@cs.duke.edu`

<sup>2</sup> Technical University of Crete, Greece. `vsam@softnet.tuc.gr`

**Abstract.** In this paper we study the external memory planar point enclosure problem: Given  $N$  axis-parallel rectangles in the plane, construct a data structure on disk (an index) such that all  $K$  rectangles containing a query point can be reported I/O-efficiently. This problem has important applications in e.g. spatial and temporal databases, and is dual to the important and well-studied orthogonal range searching problem. Surprisingly, we show that one cannot construct a linear sized external memory point enclosure data structure that can be used to answer a query in  $O(\log_B N + K/B)$  I/Os, where  $B$  is the disk block size. To obtain this bound,  $\Omega(N/B^{1-\epsilon})$  disk blocks are needed for some constant  $\epsilon > 0$ . With linear space, the best obtainable query bound is  $O(\log_2 N + K/B)$ . To show this we prove a general lower bound on the tradeoff between the size of the data structure and its query cost. We also develop a family of structures with matching space and query bounds.

## 1 Introduction

In this paper we study the external memory planar *point enclosure* problem: Given  $N$  axis-parallel rectangles in the plane, construct a data structure on disk (an index) such that all  $K$  rectangles containing a query point  $q$  can be reported I/O-efficiently. This problem is the dual of the orthogonal range searching problem, that is, the problem of storing a set of points in the plane such that the points in a query rectangle can be reported I/O-efficiently. The point enclosure problem has important applications in temporal databases; for example, if we have a database where each object is associated with a time span and a key range, retrieving all objects with key ranges containing a specific key value at a certain time corresponds to a point enclosure query. Also, in spatial databases, irregular planar objects are often represented in a data structure by their minimal bounding boxes; retrieving all bounding boxes that contain a query point, i.e. a point enclosure query, is then the first step in retrieving all objects that contain the query point. Point enclosure can also be used as part of an algorithm for finding all bounding boxes intersecting a query rectangle, since such

---

\* Supported in part by the National Science Foundation through RI grant EIA-9972879, CAREER grant CCR-9984099, ITR grant EIA-0112849, and U.S.-Germany Cooperative Research Program grant INT-0129182.

a query can be decomposed into a point enclosure query, a range query, and two segment intersection queries. While a lot of work has been done on developing worst-case efficient external memory data structures for range searching problems (see e.g. [3,17] for surveys), the point enclosure problem has only been considered in some very restricted models of computation [1,14]. In this paper we prove a general lower bound on the tradeoff between the size of an external memory point enclosure structure and its query cost, and show that this tradeoff is asymptotically tight by developing a family of structures with matching space and query bounds. Surprisingly, our results show that the point enclosure problem is harder in external memory than in internal memory.

### 1.1 Previous Work

The B-tree [13] is the most fundamental external memory data structure. It uses  $O(N/B)$  disk blocks of size  $B$  to store  $N$  elements and can be used to answer a one-dimensional range query in  $O(\log_B N + K/B)$  I/Os; an I/O is the movement of one disk block between disk and main memory. These bounds are optimal in comparison-based models of computation, and they are the bounds we would like to obtain for more complicated external memory data structure problems. In the study of such problems, a number of different lower bound models have been developed in recent years: the *non-replicating index* model [21], the *external memory pointer machine* model [25], the *bounding-volume hierarchy* model [1], and the *indexability* model [20,19]. The most general of these models is the indexability model of Hellerstein, Koutsoupias, and Papadimitriou [20,19]. To our knowledge, it captures all known external data structures for reporting problems (that is, problems where the output is a subset of the objects in the structure). In this model, the focus is on bounding the number of disk blocks containing the answers to a query  $q$ , given a bound on the number of blocks required by the data structure. The cost of computing what blocks to access to answer the query (the *search cost*) is ignored. More formally, an instance of a problem is described by a workload  $W$ , which is a simple hypergraph  $(\mathcal{O}, \mathcal{Q})$ , where  $\mathcal{O}$  is the set of  $N$  objects, and  $\mathcal{Q}$  is a set of subsets of  $\mathcal{O}$ . The elements of  $\mathcal{Q} = \{q_1, \dots, q_m\}$  are called queries. An *indexing scheme*  $\mathcal{S}$  for a given workload  $W = (\mathcal{O}, \mathcal{Q})$  is a  $B$ -regular hypergraph  $(\mathcal{O}, \mathcal{B})$ , where each element of  $\mathcal{B}$  is a  $B$ -subset of  $\mathcal{O}$  (called a block), and where  $\mathcal{O} = \cup \mathcal{B}$ . An indexing scheme  $\mathcal{S}$  can be thought of as a placement of the objects of  $\mathcal{O}$  on disk pages, possibly with redundancy. The cost of answering a query  $q$  is  $\min\{|\mathcal{B}'| \mid \mathcal{B}' \subseteq \mathcal{B}, q \subseteq \cup \mathcal{B}'\}$ , i.e., the minimum number of blocks whose union contains all objects in the query. The efficiency of an indexing scheme is measured using two parameters: its *redundancy*  $r$  and its *access overhead*  $A$ . The redundancy  $r$  is defined to be the average number of copies of an object stored in the indexing scheme, i.e.  $r = B|\mathcal{B}|/N$ , and the access overhead  $A$  is defined to be the worst-case ratio between the cost of a query and the ideal cost  $\lceil |q|/B \rceil$  (where  $|q|$  denotes the query output size). In other words, an indexing scheme with redundancy  $r$  and access overhead  $A$  occupies  $r\lceil N/B \rceil$  disk blocks and any query  $q$  is covered by at most  $A\lceil |q|/B \rceil$  disk blocks.

Following the introduction of the indexability model, a sequence of results by Koutsoupias and Taylor [22], Samoladas and Miranker [24] and Arge, Samoladas, and Vitter [6] proved a tradeoff of  $r = \Omega(\log(N/B)/\log A)$  for the (two-dimensional) orthogonal range searching problem in the model. Using this tradeoff, Arge, Samoladas and Vitter [6] proved that  $\Omega(\frac{N}{B} \frac{\log(N/B)}{\log \log_B N})$  space is needed to design an indexing scheme that answers queries in  $O(\log_B N + K/B)$  I/Os, i.e., that two-dimensional range searching is harder than the one-dimensional case. They also designed a structure with matching bounds; note that this structure works in the classical I/O-model [2] where the search cost *is* considered. A similar bound was proven in the external memory pointer machine model by Subramanian and Ramaswamy [25], and the result resembles the internal memory pointer machine case, where  $\Theta(N \log N / \log \log N)$  space is needed to obtain  $O(\log N + K)$  query cost [11]. If only  $O(N/B)$  space can be used,  $\Theta((N/B)^\epsilon)$  I/Os are needed to answer a query [6]. If exactly  $\lceil N/B \rceil$  space is allowed,  $\Theta(\sqrt{N/B})$  I/Os are needed [21].

Compared to orthogonal range searching, relatively little is known about the dual point enclosure problem. Arge and Vitter [7] designed a linear space external interval tree structure for the one-dimensional version of the problem where the data is intervals. This structure answers point enclosure queries, also called *stabbing queries*, in  $O(\log_B N + K/B)$  I/Os. The two-dimensional version of the problem we consider in this paper can be solved using the linear space R-tree [18] or its variants (see e.g. [17] for a survey); very recently Arge et al. [5] designed a variant that answers a query in worst-case  $O(\sqrt{N/B} + K/B)$  I/Os. This is optimal in the very restrictive bounding-volume hierarchy model [1]. However, in the internal memory pointer machine model a linear size data structure that can answer a query in the optimal  $O(\log N + K)$  time has been developed [10]. Thus, based on the correspondence between internal and external memory results for the range searching problem, as well as for the one-dimensional enclosure problem, one would expect to be able to develop a linear space and  $O(\log_B N + K/B)$  I/O query structure. No such structure is known.

## 1.2 Our Results

Surprisingly, in this paper we show that in the indexability model it is *not* possible to design a linear sized point enclosure indexing scheme that can answer a query in  $O(\log_B N + K/B)$  I/Os. More precisely, we show that to obtain an  $O(\log_B N + K/B)$  query bound,  $\Omega(N/B^{1-\epsilon})$  disk blocks are needed for some constant  $\epsilon > 0$ ; with linear space the best obtainable query bound is  $\Omega(\log_2 N + K/B)$ . Thus in some sense this problem is harder in external memory than in internal memory. An interesting corollary to this result is that, unlike in internal memory, an external interval tree cannot be made partially persistent without increasing the asymptotic space or query bound. Refer to Table 1 for a summary of tight complexity results for the orthogonal range search and point enclosure problems when we are restricted to linear space or a logarithmic (base 2 or  $B$ ) query bound.

**Table 1.** Summary of asymptotically tight complexity results for the orthogonal range searching and point enclosure problems in the internal memory pointer machine model and the external memory indexability model.

Range searching	Internal memory	External memory
Query with linear space	$N^\epsilon$ [9]	$(N/B)^\epsilon$ [6]
Space with log (base 2 or $B$ ) query	$N \frac{\log N}{\log \log N}$ [10,11]	$\frac{N}{B} \frac{\log(N/B)}{\log \log_B N}$ [6]

Point enclosure	Internal memory	External memory
Query with linear space	$\log_2 N$ [10]	$\log_2(N/B)$ New
Space with log (base 2 or $B$ ) query	$N$ [10]	$N/B^{1-\epsilon}$ New

In section 2 we prove our lower bounds by proving a general tradeoff between the size of a point enclosure indexing scheme and its query cost. To do so, we refine the indexability model in a way similar to [6]: We introduce parameters  $A_0$  and  $A_1$  instead of  $A$ , and require that any query  $q$  can be covered by at most  $A_0 + A_1 \lceil |q|/B \rceil$  blocks rather than  $A \lceil |q|/B \rceil$  blocks. With this refinement, we prove the lower bound tradeoff  $A_0 A_1^2 = \Omega(\log(N/B)/\log r)$ , which leads to the results mentioned above. Interestingly, if the known tradeoff for range search indexing schemes is expressed in the refined indexability model, it becomes  $r = \Omega(\log(N/B)/\log(A_0 \cdot A_1))$ . Thus, the tradeoffs of the dual range search and point enclosure indexing problems are symmetric with respect to  $r$  and  $A_0, A_1$ .

In Section 3 we show that our lower bound for the tradeoff is asymptotically tight in the most interesting case  $A_1 = O(1)$  by developing a family of external memory data structures (where the search cost is considered) with optimal space and query cost. More precisely, we describe a structure that, for any  $2 \leq r \leq B$ , uses  $O(rN/B)$  disk blocks and answers queries in  $O(\log_r(N/B) + K/B)$  I/Os. The structure can be constructed in  $O(r \frac{N}{B} \log_{M/B} \frac{N}{B})$  I/Os, where  $M$  is the size of main memory.

## 2 Lower Bounds

### 2.1 Refined Redundancy Theorem

The Redundancy Theorem of [24,19] is the main tool in most indexability model lower bound results. We develop a version of the theorem for the refined indexability model, that is, the model where the access overhead  $A$  has been replaced by two parameters  $A_0$  and  $A_1$ , such that a query is required to be covered by at most  $A_0 + A_1 \lceil |q|/B \rceil$  blocks.

**Theorem 1 (Redundancy Theorem [24,19]).** *For a workload  $W = (\mathcal{O}, \mathcal{Q})$ , where  $\mathcal{Q} = \{q_1, q_2, \dots, q_m\}$ , let  $\mathcal{S}$  be an indexing scheme with access overhead  $A \leq \sqrt{B}/4$  such that for any  $1 \leq i, j \leq m, i \neq j$ :*

$$|q_i| \geq B/2 \quad \text{and} \quad |q_i \cap q_j| \leq \frac{B}{16A^2},$$

then the redundancy of  $\mathcal{S}$  is bounded by

$$r \geq \frac{1}{12N} \sum_{i=1}^m |q_i|.$$

We extend this theorem to the refined indexability model as follows.

**Theorem 2 (Refined Redundancy Theorem).** *For a workload  $W = (\mathcal{O}, \mathcal{Q})$ , where  $\mathcal{Q} = \{q_1, q_2, \dots, q_m\}$ , let  $\mathcal{S}$  be an indexing scheme with access overhead  $(A_0, A_1)$  with  $A_1 \leq \sqrt{B}/8$  such that for any  $1 \leq i, j \leq m, i \neq j$ :*

$$|q_i| \geq BA_0 \quad \text{and} \quad |q_i \cap q_j| \leq \frac{B}{64A_1^2},$$

then the redundancy of  $\mathcal{S}$  is bounded by

$$r \geq \frac{1}{12N} \sum_{i=1}^m |q_i|.$$

*Proof.* Since for any  $i$ ,  $|q_i| \geq BA_0$ ,  $q_i$  is required to be covered by at most  $A_0 + A_1 \lceil |q_i|/B \rceil \leq 2A_1 \lceil |q_i|/B \rceil$  blocks. If we set  $A = 2A_1$ ,  $\mathcal{S}$  is an indexing scheme that covers each query of  $\mathcal{Q}$  by  $A \lceil |q|/B \rceil$  blocks. Then applying Theorem 1 leads to the desired result.

## 2.2 Lower Bound for Point Enclosure

In order to apply Theorem 2 to the point enclosure problem we need to design a workload  $W = (\mathcal{O}, \mathcal{Q})$  such that each query is sufficiently large but the intersection of any two is relatively small. To do so we use the point set called a *Fibonacci lattice*, which was also used in previous results on orthogonal range searching [22,6,19].

**Definition 1 ([23]).** *The Fibonacci lattice  $F_m$  is a set of two-dimensional points defined by  $F_m = \{(i, if_{k-1} \bmod m) | i = 0, 1, \dots, m-1\}$ , where  $m = f_k$  is the  $k$ th Fibonacci number.*

The following property of a Fibonacci lattice will be crucial in our lower bound tradeoff proof.

**Lemma 1 ([16]).** *For the Fibonacci lattice  $F_m$  and for  $\alpha \geq 0$ , any rectangle with area  $\alpha m$  contains between  $\lfloor \alpha/c_1 \rfloor$  and  $\lceil \alpha/c_2 \rceil$  points, where  $c_1 \approx 1.9$  and  $c_2 \approx 0.45$ .*

Let  $m = \lambda \frac{\alpha N}{BA_0}$ , where  $\alpha$  is a parameter to be determined later, and  $1 \leq \lambda < 2$  is chosen such that  $m$  is a Fibonacci number. The idea is to use the points in  $F_m$  as queries to form  $\mathcal{Q}$  and construct approximately  $N$  rectangles as objects to form  $\mathcal{O}$ ; in the previous range searching tradeoff the roles of the points and rectangles

were reversed [6]. We may not construct exactly  $N$  rectangles but the number will be between  $\lambda N$  and  $4\lambda N$ , so this will not affect our asymptotic result.

We construct rectangles of dimensions  $\alpha t^i \times m/t^i$ , where  $t = (m/\alpha)^{1/(BA_0)}$  and  $i = 1, \dots, BA_0$ . For each choice of  $i$ , the rectangles are constructed in a tiling fashion on  $F_m$  until they cross the boundary. In this way, between  $\frac{m^2}{\alpha m} = \frac{\lambda N}{BA_0}$  and  $4\frac{\lambda N}{BA_0}$  rectangles are constructed on each layer (each  $i$ ), for a total of  $\Theta(N)$  rectangles.

Since each point is covered by  $BA_0$  rectangles, one from each layer, the first condition of Theorem 2 is satisfied. For the second one, consider any two different query points  $q_1$  and  $q_2 \in F_m$ , and let  $x$  and  $y$  be the differences between their  $x$  and  $y$ -coordinates, respectively. Since the rectangle with  $q_1$  and  $q_2$  as corners contains at least two points, namely  $q_1$  and  $q_2$  themselves, by Lemma 1, we have  $xy \geq c_2 m$ . We now look at how many rectangles can possibly cover both  $q_1$  and  $q_2$ . For a rectangle with dimension  $\alpha t^i \times m/t^i$  to cover both points, we must have  $\alpha t^i \geq x$  and  $m/t^i \geq y$ , or equivalently,  $x/\alpha \leq t^i \leq m/y$ . rectangle for each  $i$  that can cover both points, thus,  $|q_1 \cap q_2|$  is at most

$$\left\lceil \log_t \frac{\alpha m}{xy} \right\rceil \leq \left\lceil \frac{\log(\alpha/c_2)}{\log t} \right\rceil = \left\lceil BA_0 \frac{\log(\alpha/c_2)}{\log(m/\alpha)} \right\rceil \leq 1 + BA_0 \frac{\log(\alpha/c_2)}{\log(\lambda N/(BA_0))}.$$

The second condition holds as long as

$$\frac{1}{B} + A_0 \frac{\log(\alpha/c_2)}{\log(\lambda N/(BA_0))} \leq \frac{1}{64A_1^2}. \quad (1)$$

On the other hand, when (1) is satisfied, by Theorem 2, we have

$$r \geq \frac{1}{12} \frac{mBA_0}{4\lambda N} = \frac{\alpha}{48}.$$

Therefore, any combination of  $\alpha$ ,  $A_0$  and  $A_1$  that satisfy (1) constitutes a lower bound on the tradeoff of  $r$ ,  $A_0$  and  $A_1$ . When  $A_0$  and  $A_1$  are small enough, namely  $A_0 < (N/B)^{1-\delta}$  where  $0 < \delta < 1$  is some constant, and  $A_1 < \sqrt{B/128}$ , we can simplify (1) to obtain the following:

**Theorem 3.** *Let  $\mathcal{S}$  be a point enclosure indexing scheme on the Fibonacci workload. If  $A_0 \leq (\frac{N}{B})^{1-\delta}$  for any fixed  $0 < \delta < 1$ , and  $A_1 \leq \sqrt{B/128}$ , then its redundancy  $r$  and access overhead  $(A_0, A_1)$  must satisfy*

$$A_0 A_1^2 \geq \frac{\delta}{128} \frac{\log(N/B)}{7 + \log r} = \Omega\left(\frac{\log(N/B)}{\log r}\right).$$

The Fibonacci lattice is only one of many low-discrepancy point sets [23] we could have used, but none of them would improve the result of Theorem 3 by more than a constant factor. We note that the above proof technique could also have been used to obtain a tradeoff in the original indexability model. However, in that case we would obtain a significantly less informative tradeoff of  $A = \Omega(\sqrt{\log(N/B)/\log r})$ .

### 2.3 Tradeoff Implications

*Query cost*  $\log_B N$ . In indexing schemes with an  $O(\log_B N + K/B)$  query cost we have  $A_0 = O(\log_B N)$  and  $A_1 = O(1)$ . Assuming that  $A_0 \leq c_0 \log_B \frac{N}{B}$ , we have

$$c_0 \frac{\log(N/B)}{\log B} A_1^2 \geq \frac{\delta}{128} \frac{\log(N/B)}{7 + \log r},$$

which is

$$\log r \geq \frac{\delta}{128 c_0 A_1^2} \log B - 7, \text{ or } r \geq \frac{1}{128} B^{\delta/(128 c_0 A_1^2)}.$$

Thus any such indexing scheme must have redundancy at least  $\Omega(B^\epsilon)$ , for some small constant  $\epsilon$ .

**Corollary 1.** *Any external memory data structure for the point enclosure problem that can answer a query in  $O(\log_B N + K/B)$  I/Os in the worst case must use  $\Omega(N/B^{1-\epsilon})$  disk blocks for some constant  $\epsilon > 0$ .*

*Linear space.* In linear space indexing schemes  $r$  is a constant. If we also want  $A_1$  to be a constant,  $A_0$  has to be  $\Omega(\log \frac{N}{B})$  by Theorem 3. As mentioned, this is a rather surprising result, since linear space structures with the optimal  $O(\log N + K)$  query time exist in internal memory [10].

**Corollary 2.** *Any linear sized external memory data structure for the point enclosure problem that answers a query in  $f(N) + O(K/B)$  I/Os in the worst case must have  $f(N) = \Omega(\log_2 \frac{N}{B})$ .*

Note however that our lower bound does *not* rule out linear size indexes with a query cost of for example  $O(\log_B N + \sqrt{\log B} \frac{K}{B})$ .

*Persistent interval tree.* An interesting consequence of the above linear space result is that, unlike in internal memory, we cannot make the external interval tree [7] partially persistent [15] without either increasing the asymptotic space or query bound. Details will appear in the full paper.

## 3 Upper Bounds

In this section, we develop a family of external memory data structures with space and query bounds that match the lower bound tradeoff of Theorem 3. In Section 3.1 we first develop a structure that uses  $O(rN/B)$  disk blocks and answers queries in  $O(\log_B N \cdot \log_r(N/B) + K/B)$  I/Os for any  $2 \leq r \leq B$ . In Section 3.2 we then discuss how to improve the query bound to obtain the  $O(\log_r(N/B) + K/B)$  bound matching Theorem 3. These bounds are measured in the classical I/O model [2], where the search cost is considered, and where space used to store auxiliary information (such as e.g. “directories” or “internal nodes”) is also counted when bounding the sizes of the structures.

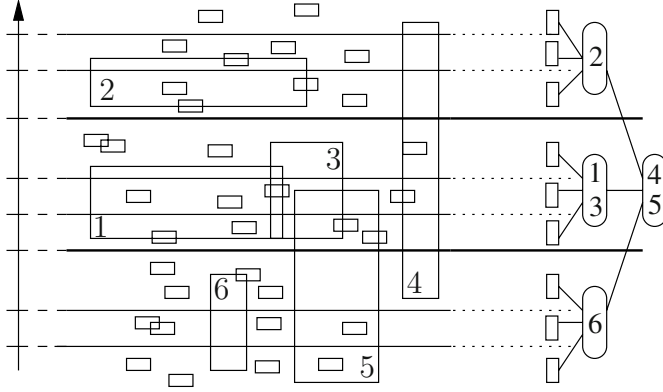


Fig. 1. Partitioning of rectangles among nodes in a 3-ary base tree.

### 3.1 Basic Point Enclosure Structure

*Base tree  $T$ .* Let  $S$  be a set of  $N$  axis-parallel rectangles in the plane. Our structure for answering point enclosure queries on  $S$  is similar to an internal memory point enclosure structure due to Chazelle [10]. Intuitively, the idea is to build an  $r$ -ary base tree<sup>1</sup>  $T$  by first dividing the plane into  $r \leq B$  horizontal *slabs* with approximately the same number of rectangle corners in each slab, and then recursively construct a tree on the rectangles completely contained in each of the slabs. The rectangles that cross one or more slab boundaries are stored in a secondary structure associated with the root of  $T$ . The recursion ends when the slabs contain at most  $4B$  rectangle corners each. Refer to Figure 1.

The base tree  $T$  can relatively easily be constructed and the rectangles distributed to the internal nodes of  $T$  in  $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$  I/Os (the number of I/Os needed to sort  $N$  elements): We first sort the rectangle corners by  $y$ -coordinate. Then we construct the top  $\Theta(\log_r \frac{M}{B})$  levels of the tree, that is,  $O(M/B)$  nodes, in  $O(N/B)$  I/Os. Finally, we distribute the remaining (sorted) rectangles to the leaves of the constructed tree and recursively construct the corresponding subtrees. Details will appear in the full version of this paper.

Now let  $N_v$  be the number of rectangles associated with an internal node  $v$  in  $T$ . Below we describe how the secondary structure associated with  $v$  uses  $O(rN_v/B + r)$  blocks and can be constructed in  $O(\frac{rN_v}{B} \log_{M/B} \frac{N_v}{B})$  I/Os. Since there are  $O(N/(Br))$  internal nodes, and since each rectangle is stored in exactly one leaf or internal node, this means that after distributing the rectangles to nodes of  $T$ , all the secondary structures use  $O(rN/B)$  blocks and they can be constructed in  $O(\frac{rN}{B} \log_{M/B} \frac{N}{B})$  I/Os.

*Answering a query on  $T$ .* To answer a query  $q = (x_q, y_q)$  on  $T$  we simply query the secondary structure of the root  $v$  of  $T$ , and then recursively query the tree

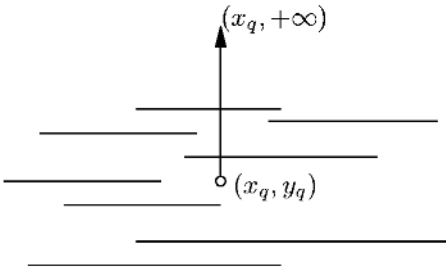
<sup>1</sup> The fanout of the root of the base tree may be less than  $r$  to make sure that we have  $O(N/B)$  nodes in the tree.



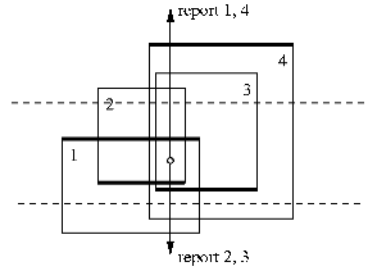
rooted in the node corresponding to the horizontal slab containing  $q$ . When we reach a leaf, we simply load the at most  $B$  rectangles in the leaf and report all rectangles that contain  $q$ . Below we describe how a query on the secondary structure of  $v$  can be performed in  $O(\log_B N + K_v/B)$  I/Os, where  $K_v$  is the number of reported rectangles. Thus a query is answered in  $O(\log_B N \cdot \log_r \frac{N}{B} + K/B)$  I/Os overall.

*Secondary structures.* All that remains is to describe the secondary structure used to store the  $N_v$  rectangles associated with an internal node  $v$  of  $T$ . The structure actually consists of  $2r$  small structures, namely two structures for each of the  $r$  horizontal slabs associated with  $v$ . The first structure  $\Psi_t^i$  for slab  $i$  contains the top (horizontal) sides of all rectangles with top side in slab  $i$ , as well as all rectangles that completely span slab  $i$ ; the second structure  $\Psi_b^i$  contains the bottom (horizontal) sides of all rectangles with bottom side in slab  $i$ . The structure  $\Psi_t^i$  supports *upward ray-shooting queries* from a point  $q = (x_q, y_q)$ , that is, it can report all segments intersected by the vertical half-line from  $(x_q, y_q)$  to  $(x_q, +\infty)$ . Refer to Figure 2. Similarly,  $\Psi_b^i$  supports *downward ray-shooting queries*. Thus, it is easy to see that to answer a point enclosure query  $q = (x_q, y_q)$  in  $v$ , all we need to do is to query the  $\Psi_t^i$  and  $\Psi_b^i$  structures of the slab  $i$  containing point  $(x_q, y_q)$ . Refer to Figure 3.

All  $\Psi_t$  and  $\Psi_b$  structures are implemented in the same basic way, using a (partially) persistent B-tree [8,4]. A persistent B-tree uses  $O(N/B)$  blocks, supports updates in  $O(\log_B N)$  I/Os in the current version of the structure as normal B-trees, and answers range queries in  $O(\log_B N + K/B)$  I/Os in all the previous versions of the structure. In these bounds,  $N$  is the number of updates performed on it. To build  $\Psi_t$  we simply imagine sweeping the plane with a vertical line from  $-\infty$  to  $+\infty$ , while inserting the  $y$ -coordinate  $y_1$  of a horizontal segment  $(x_1, y_1, x_2)$  when its left endpoint  $x_1$  is reached, and deleting it again when its right endpoint  $x_2$  is reached. An upward ray-shooting query  $q = (x_q, y_q)$  can then be answered in  $O(\log_B N + K_v/B)$  I/Os as required, simply by performing a range query  $(y_q, +\infty)$  on the structure we had when the sweep-line was at  $x_q$ .  $\Psi_b$  is constructed in a symmetric manner.



**Fig. 2.** An upward ray-shooting query.



**Fig. 3.** Answering a point enclosure query.

The top horizontal segment of each of the  $N_v$  rectangles associated with  $v$  can be stored in  $r$   $\Psi_t$  structures, while the bottom segments are stored in exactly one  $\Psi_b$ . Therefore, since a  $\Psi_t$  or  $\Psi_b$  structure on  $L$  segments uses  $O(L/B)$  blocks and can be constructed in  $O(\frac{L}{B} \log_{M/B} \frac{L}{B})$  I/Os [26], the  $2r$  structures in  $v$  use  $O(rN_v/B + r)$  blocks and can be constructed in  $O(\frac{rN_v}{B} \log_{M/B} \frac{N_v}{B})$  I/Os overall.

**Theorem 4.** *A set of  $N$  axis-parallel rectangles in the plane can be stored in an external memory data structure that uses  $O(rN/B)$  blocks, such that a point enclosure query can be answered in  $O(\log_B N \cdot \log_r \frac{N}{B} + K/B)$  I/Os, for any  $2 \leq r \leq B$ . The structure can be constructed using  $O(r \frac{N}{B} \log_{M/B} \frac{N}{B})$  I/Os.*

### 3.2 Improved Point Enclosure Structure

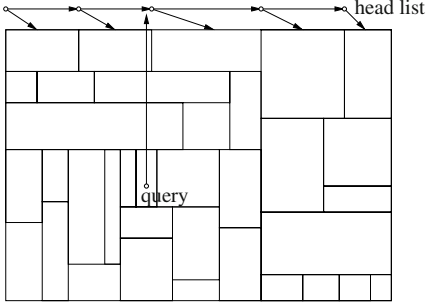
In this section we sketch how to improve the  $O(\log_B N \cdot \log_r(N/B) + K/B)$  query bound of the structure described in the previous section to  $O(\log_r(N/B) + K/B)$ . First note that the extra  $\log_B N$ -term was a result of the query procedure using  $O(\log_B N)$  I/Os to search a secondary structure (a  $\Psi_t$  and a  $\Psi_b$  structure) on each level of the base tree  $T$ . Since all of these structures are queried with the same query point  $q$ , it is natural to use fractional cascading [12] to reduce the overall search cost to  $O(\log_B N + \log_r(N/B) + K/B) = O(\log_r(N/B) + K/B)$ . However, in order to do so we need to modify the  $\Psi_t$  and  $\Psi_b$  structures slightly. Below we first discuss this modification and then sketch how fractional cascading can be applied to our structure. We will only consider the  $\Psi_t$  structure since the  $\Psi_b$  structure can be handled in a similar way.

*Modified secondary structure.* The modification of the  $\Psi_t$  structure is inspired by the so-called *hive-graph* technique of Chazelle [10]. In order to describe it, we need the following property of a  $\Psi_t$  structure (that is, of a persistent B-tree), which follows easily from the discussions in [8].

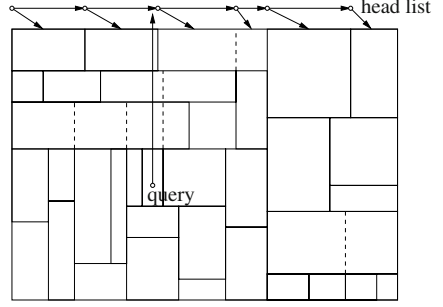
**Lemma 2.** *Each of the  $O(L/B)$  leaves of a persistent B-tree  $\Psi_t$  on  $L$  segments defines a rectangular region in the plane and stores the (at most)  $B$  segments intersecting this region. The regions of all the leaves define a subdivision of the minimal bounding box of the segments in  $\Psi_t$  and no two regions overlap. An upward ray-shooting query on  $\Psi_t$  visits  $O(1 + K/B)$  leaves.*

By Lemma 2 an upward ray shooting query  $q = (x_q, y_q)$  on a  $\Psi_t$  in node  $v$  will visit  $O(1 + K_v/B)$  regions (leaves) in the subdivision induced by the leaves. Now suppose we have links from any region (leaf) to its downward neighbors. Given the highest region that contains  $x_q$  we could then follow these links downward to answer the query. To find the relevant highest regions, we introduce a *head list* consisting of the left  $x$ -coordinates of the  $O(L/B)$  highest regions in sorted order, where each entry also has a pointer to the corresponding region/leaf. The top region can then be found simply by searching the head list. Refer to Figure 4.

There is one problem with the above approach, namely that a region may have more than  $B$  downward neighbors (so that following the right link may



**Fig. 4.** The leaves of a persistent B-tree defines a rectangular subdivision.



**Fig. 5.** Reorganizing the leaves in the persistent B-tree ( $B = 2$ )

require more than one I/O). We therefore modify the subdivision slightly: We imagine sweeping a horizontal line from  $-\infty$  to  $\infty$  and every time we meet the bottom edge of a region with more than  $B$  downward links, we split it into smaller regions with  $\Theta(B)$  downward links each. We construct a new leaf for each of these regions, all containing the relevant segments from the original region (the segments intersecting the new region). Refer to Figure 5. Chazelle showed that the number of new regions (leaves) introduced during the sweep is  $O(L/B)$  [10], so the modified subdivision still consists of  $O(L/B)$  regions. It can also be constructed in  $O(\frac{L}{B} \log_{M/B} \frac{L}{B})$  I/Os. Due to lack of space, we omit the details. They will appear in the full paper.

**Lemma 3.** *A modified  $\Psi_t$  structure on  $L$  segments consists of  $O(L/B)$  blocks and a head list of  $O(L/B)$   $x$ -coordinates, such that after locating  $x_q$  in the head list, an upward ray-shooting query  $q = (x_q, y_q)$  can be answered in  $O(1 + K/B)$  I/Os. The structure can be constructed in  $O(\frac{L}{B} \log_{M/B} \frac{L}{B})$  I/Os.*

*Fractional cascading.* We are now left with the task of locating  $x_q$  in the head list of each  $\Psi_t$  structure that we visit along a root-to-leaf path in the base tree  $T$ . This repetitive searching problem can be solved efficiently by adapting the fractional cascading technique [12] to the external memory setting, such that after locating  $x_q$  in the first head list, finding the position of  $x_q$  in each of the subsequent head lists only incurs constant cost. Again, due to space limitations we omit the details. In the full version of this paper we show how the search cost of our structure can be reduced to  $O(\log_r \frac{N}{B} + K/B)$  I/Os while maintaining the space and construction bounds; this leads to our main result.

**Theorem 5.** *A set of  $N$  axis-parallel rectangles in the plane can be stored in an external memory data structure that uses  $O(rN/B)$  blocks such that a point enclosure query can be answered in  $\Theta(\log_r \frac{N}{B} + K/B)$  I/Os, for any  $2 \leq r \leq B$ . The structure can be constructed with  $O(r \frac{N}{B} \log_{M/B} \frac{N}{B})$  I/Os.*

## References

1. P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammer, and H. J. Haverkort. Box-trees and R-trees with near-optimal query time. In *Proc. ACM Symposium on Computational Geometry*, pages 124–133, 2001.
2. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
3. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.
4. L. Arge, A. Danner, and S.-H. Teh. I/O-efficient point location using persistent B-trees. In *Proc. Workshop on Algorithm Engineering and Experimentation*, 2003.
5. L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. In *Proc. SIGMOD International Conference on Management of Data*, 2004.
6. L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symposium on Principles of Database Systems*, pages 346–357, 1999.
7. L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.
8. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.
9. J. L. Bentley. Multidimensional divide and conquer. *Communications of the ACM*, 23(6):214–229, 1980.
10. B. Chazelle. Filtering search: a new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.
11. B. Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM*, 37(2):200–212, Apr. 1990.
12. B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.
13. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
14. M. de Berg, J. Gudmundsson, M. Hammar, and M. Overmars. On R-trees with low stabbing number. In *Proc. European Symposium on Algorithms*, pages 167–178, 2000.
15. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.
16. A. Fiat and A. Shamir. How to find a battleship. *Networks*, 19:361–371, 1989.
17. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
18. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD International Conference on Management of Data*, pages 47–57, 1984.
19. J. Hellerstein, E. Koutsoupias, D. Miranker, C. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *Journal of ACM*, 49(1), 2002.
20. J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *Proc. ACM Symposium on Principles of Database Systems*, pages 249–256, 1997.
21. K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proc. International Conference on Database Theory, LNCS 1540*, pages 257–276, 1999.

22. E. Koutsoupias and D. S. Taylor. Tight bounds for 2-dimensional indexing schemes. In *Proc. ACM Symposium on Principles of Database Systems*, pages 52–58, 1998.
23. J. Matoušek. *Geometric Discrepancy*. Springer, 1999.
24. V. Samoladas and D. Miranker. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proc. ACM Symposium on Principles of Database Systems*, pages 44–51, 1998.
25. S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 378–387, 1995.
26. J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. International Conference on Very Large Databases*, pages 406–415, 1997.

# Maximizing Throughput in Multi-queue Switches

Yossi Azar\* and Arik Litichevsky

School of Computer Science, Tel Aviv University, Tel Aviv, 69978, Israel.  
{azar,litiche}@tau.ac.il

**Abstract.** We study a basic problem in Multi-Queue switches. A switch connects  $m$  input ports to a single output port. Each input port is equipped with an incoming FIFO queue with bounded capacity  $B$ . A switch serves its input queues by transmitting packets arriving at these queues, one packet per time unit. Since the arrival rate can be higher than the transmission rate and each queue has limited capacity, packet loss may occur as a result of insufficient queue space. The goal is to maximize the number of transmitted packets. This general scenario models most current networks (e.g., IP networks) which only support a “best effort” service in which all packet streams are treated equally. A 2-competitive algorithm for this problem was designed in [4] for arbitrary  $B$ . Recently, a  $\frac{17}{9} \approx 1.89$ -competitive algorithm was presented for  $B > 1$  in [2]. Our main result in this paper shows that for  $B$  which is not too small our algorithm can do better than 1.89, and approach a competitive ratio of  $\frac{e}{e-1} \approx 1.58$ .

## 1 Introduction

**Overview:** Switches are a fundamental part of most networks. Networks use switches to route packets arriving at input ports to an appropriate output port, so that the packets will reach their desired destination. Since the arrival rate of packets can be much higher than the transmission rate, each input port is equipped with an incoming queue with bounded capacity. Considering that on the one hand, traffic in networks, which has increased steadily during recent years, tends to fluctuate, and on the other hand, incoming queues have limited space, packet loss is becoming more of a problem. As a result, over the recent years, considerable research has been carried out in the design and analysis of various queue management policies.

We model the problem of maximizing switch throughput as follows. A switch has  $m$  incoming FIFO queues and one output port. At each time unit, new packets may arrive at the queues, each packet belonging to a specific input queue. A packet can only be stored in its input queue if there is enough space. Since the  $m$  queues have bounded capacity, packet loss may occur. All packets have the same value, i.e. all packets are equally important. At each time unit,

---

\* Supported by Israeli Ministry of industry and trade and Israel Science Foundation.

the switch can select a non-empty queue and transmit a packet from the head of that queue. The goal is to maximize the number of transmitted packets.

This model is worth studying, since the majority of current networks (most notably, IP networks) do not yet integrate full QoS capabilities. Traditionally, similar problems were analyzed while assuming either some constant structure of the sequence of arriving packets, or a specific distribution of the arrival rates (see e.g. [7,15]). We, in contrast, avoid any a priori assumption on the input, and therefore use competitive analysis to evaluate the performance of online algorithms to the optimal off-line solution that knows the entire sequence in advance. Specifically, the competitive ratio of an online algorithm is the supremum, taken over all finite sequences, of the ratio between the online throughput and optimal throughput on the same input.

In [4] Azar and Richter showed that any deterministic work-conserving algorithm, i.e. transmits a packet in each unit of time if not all queues are empty, is 2-competitive. They also showed an  $\frac{e}{e-1}$ -competitive randomized algorithm. Recently, Albers and Schmidt [2] introduced a  $\frac{17}{9} \approx 1.89$ -competitive algorithm for  $B > 1$ . In this paper we show that for the case where  $B$  is larger than  $\log m$  our algorithm approaches a competitive ratio of  $\frac{e}{e-1} \approx 1.58$ , which is significantly better than 1.89.

### Our results:

- Our main contribution is an  $\frac{e}{e-1} \approx 1.58$ -competitive algorithm for the switch throughput problem, for any large enough  $B$ . Specifically, we show a *fractional* algorithm for the switch throughput problem, i.e. one that can insert *fractions* of packets into each queue if there is sufficient space and transmit a fraction from each queue of total of at most 1, with competitiveness of  $\frac{e}{e-1}$ . Then we transform our *fractional* algorithm into a discrete algorithm, i.e. one that can insert and transmit integral packets, with competitiveness of  $\frac{e}{e-1}(1 + \frac{\lfloor H_m \rfloor + 1}{B})$ , where  $H_m = \sum_{i=1}^m \frac{1}{i} \approx \ln m$  is the  $m$ th harmonic number.
- We use two tools which are of their own interest :
  - We consider the online unweighted fractional matching problem in a bipartite graph. In this problem we have a bipartite graph  $G = (R, S, E)$  where  $S$  and  $R$  are the disjoint vertex sets and  $E$  is the edge set. At step  $i$ , vertex  $r_i \in R$  together with all of its incident edges, arrives online. The online algorithm can match fractions of  $r_i$  to various adjacent vertices  $s_j \in S$ . We prove an upper bound of  $\frac{e}{e-1}$  on the competitive ratio of a natural online “water level” matching algorithm.
  - We present a generic technique to transform any *fractional* algorithm for the switch throughput problem into a discrete algorithm. Specifically, we show that given a  $c$ -competitive online fractional algorithm for the switch throughput problem, we can construct an online discrete algorithm with competitiveness of  $c(1 + \frac{\lfloor H_m \rfloor + 1}{B})$ , for the switch throughput problem.

### Side results:

- We show a lower bound of  $\frac{e}{e-1} - \Theta(\frac{1}{m})$  on the competitive ratio of any online unweighted fractional matching algorithm in a bipartite graph. Thus,

we prove that our “water level” matching algorithm is optimal, up to an additive smaller than 2 for the size of the matching. In addition, this lower bound yields a lower bound of  $\frac{e}{e-1} - \Theta(\frac{1}{m})$  on the competitive ratio of any randomized discrete algorithm for the online unweighted matching problem in a bipartite graph. Therefore, we slightly improve the asymptotic lower bound of Karp *et al.* [10] with a non-asymptotic lower bound.

- We consider the online b-matching problem. In the online b-matching problem we have a bipartite graph  $G = (R, S, E)$ , where  $r_i \in R$  can be matched to a *single* adjacent vertex  $s_j \in S$  such that every  $s_j$  can be matched up to  $b$  times. We introduce a more compact analysis for the upper bound of the competitive ratio obtained by algorithm *Balance* for the b-matching problem that was studied by Kalyanasundaram and Pruhs [9], using a similar technique to the one which was used in our analysis of the “water level” algorithm.

**Our techniques:** We start by studying the online fractional unweighted matching problem in a bipartite graph. We introduce a natural “water level” algorithm and obtain the competitiveness of  $\frac{e}{e-1}$ . We next construct an *online* reduction from the fractional switch throughput problem to the problem of finding a maximum fractional matching in a bipartite graph. Thus, we obtain a  $\frac{e}{e-1}$ -competitive algorithm, for the fractional switch problem, which emulates the fractional matching constructed in the bipartite graph. Next, we present a generic technique to transform any fractional algorithm for the switch throughput problem into a discrete algorithm. Specifically, we present an algorithm with queues larger than  $B$  that maintains a running simulation of the fractional algorithm in a relaxed fractional model and transmits from the queue with the largest overload with respect to the simulation. We then transform this algorithm into an algorithm with queues of size  $B$ .

**Related results for the switch throughput problem with unit value:**

The online problem of maximizing switch throughput has been studied extensively during recent years. For the unit value scheduling, Azar and Richter [4] showed that any deterministic work-conserving algorithm, i.e. one that transmits a packet in each unit of time if not all queues are empty, is 2-competitive. In addition they showed a lower bound of  $2 - \frac{1}{m}$  for the case where  $B = 1$ . For arbitrary  $B$  they showed a lower bound of  $1.366 - \Theta(\frac{1}{m})$ . They also considered randomized online algorithms and presented an  $\frac{e}{e-1}$ -competitive randomized algorithm. For  $B = 1$ , they showed a lower bound of  $1.46 - \Theta(\frac{1}{m})$  on the performance of every randomized online algorithm. Recently, a  $\frac{17}{9} \approx 1.89$ -competitive deterministic algorithm was presented by Albers and Schmidt [2], for  $B \geq 2$ . For the case  $B = 2$  they showed that their algorithm is optimal. They also showed that any greedy algorithm is at least  $2 - \frac{1}{B}$ -competitive for any  $B$  and large enough  $m$ . In addition, they showed a lower bound of  $\frac{e}{e-1}$  on the competitive ratio of any deterministic online algorithm and a lower bound of 1.4659 on the performance of every randomized online algorithm, for any  $B$  and large enough  $m$ . We note that the lower bound obtained by Albers and Schmidt does not apply to  $B > \log m$  considered in our paper.



**Related results for the switch throughput problem with values:** For the case where packets have values, and the goal is to maximize the total value of the transmitted packets, there are results for a single queue and for a multi-queue switch in [1,3,12,14,11,4,5]. In particular, for the switch throughput problem there is a 3-competitive algorithm for the preemptive case, and a logarithmic in the range value competitive algorithm for the non-preemptive case.

**Related results for the cost version:** Koga [13], Bar-Noy *et al.* [6], and Chrobak *et al.* [8] showed a  $\Theta(\log m)$ -competitive algorithm for the problem of minimizing the length of the longest queue in a switch. In their model, queues are unbounded in size, and hence packets are not lost.

**Related results for the online unweighted matching problem in a bipartite graph:** In the online unweighted matching problem we have a bipartite graph  $G = (R, S, E)$  where  $S$  and  $R$  are the disjoint vertex sets and  $E$  is the edge set. At step  $i$ , vertex  $r_i \in R$  together with all of its incident edges, arrives online. The online algorithm can match vertex  $r_i \in R$  to an adjacent vertex  $s_i \in S$ , and the goal is to maximize the number of matched vertices. Karp *et al.* [10] observed that in the integral unweighted matching problem in a bipartite graph any deterministic algorithm, which never refuses to match if possible, is 2-competitive and no deterministic algorithm can be better than 2-competitive. In addition, they introduced a randomized algorithm *RANKING* with a competitive ratio of  $\frac{e}{e-1} - o(1)$  and proved a lower bound of  $\frac{e-1}{e} - o(1)$ , thus obtaining the optimality of *RANKING*, up to lower order terms. We improve the lower order terms in their lower bound. The online unweighted matching problem was generalized by Kalyanasundaram and Pruhs in [9] to the  $b$ -matching problem. In this model each vertex  $r_i \in R$  can be matched to a *single* adjacent vertex  $s_j \in S$  such that every  $s_j$  can be matched up to  $b$  times. They showed an optimal online algorithm *Balance* with competitiveness of  $\frac{(1+\frac{1}{b})^b}{(1+\frac{1}{b})^b-1}$ .<sup>1</sup>

**Paper structure:** Section 2 includes formal definitions and notation. In Section 3 we consider the online unweighted fractional matching and obtain an upper bound for a natural online algorithm. We address the fractional version of the switch throughput problem in Section 4 and obtain an upper bound for this problem. In Section 5 we present our general technique for the discretization of any fractional algorithm for the switch throughput problem and obtain an upper bound for the outcome algorithm. Open problems are presented in Section 6.

## 2 Problem Definition and Notations

We model the switch throughput maximization problem as follows. We are given a switch with  $m$  FIFO input queues, where queue  $i$  has size  $B_i$ , and one output port. Packets arrive online at the queues, every packet belonging to a specific input queue. All packets are of equal size and value. We denote the online finite packet sequence by  $\sigma$ . Initially, all  $m$  queues are empty. Each time unit is divided into two phases: in the *arrival* phase a set of packets arrives at specific input

<sup>1</sup> In [9] and [10] the authors used a smaller than 1 definition for the competitive ratio.

queues and may be inserted into the queues if there is sufficient place. Remaining packets must be discarded. In the *transmission* phase the switching algorithm may select one of the non-empty queues, if such exists, and transmit the packet at the head of that queue. The goal is to maximize the number of transmitted packets. We consider the non-preemptive model, in which stored packets cannot be discarded from the queues. We note that in the unit value model the admission control question, i.e. which packet to accept, is simple, since there is no reason to prefer one packet over the other. Therefore, it is enough to focus on the scheduling problem.

For a given time  $t$ , we use the term *load* of queue  $i$  to refer to the number of packets residing in that queue, at that time. For a given time  $t$ , the term *space* of queue  $i$  is used to refer to its size minus its load, i.e. how many additional packets can be assigned to queue  $i$ , at that time. Given an online switching algorithm  $A$  we denote by  $A(\sigma)$  the value of  $A$  given the sequence  $\sigma$ . We denote the optimal (offline) algorithm by  $OPT$ , and use similar notation for it. A deterministic online algorithm  $A$  is  $c$ -competitive for all maximization online problems described in this paper (the unweighted fractional matching problem in a bipartite graph, b-matching problem in a bipartite graph, and switch throughput maximization problem fractional version as well as discrete version) iff for every instance of the problem and every packet sequence  $\sigma$  we have:  $OPT(\sigma) \leq c \cdot A(\sigma)$ .

We prove our upper bound by assuming that all queues in the switch are of equal size  $B$ . We note that this is done for simplicity of notation only. The algorithms we present remain the same when queues have different sizes, where in our upper bound,  $B$  will be replaced by  $\min_i \{B_i\}$ .

### 3 Online Unweighted Fractional Matching in a Bipartite Graph

Our switching algorithm is based on an online fractional matching algorithm. Thus we start by considering the online unweighted matching problem in a bipartite graph, which is defined as follows. Consider an online version of the maximum bipartite matching on a graph  $G = (R, S, E)$ , where  $S$  and  $R$  are the disjoint vertex sets and  $E$  is the edge set. We refer to set  $R$  as the requests and refer to set  $S$  as the servers. The objective is to match a request to a server. At step  $i$ , vertex  $r_i \in R$  together with all of its incident edges, arrives online. The response of  $A$  can be either to reject  $r_i$ , or to irreversibly match it to an unmatched vertex  $s_j \in S$  adjacent to  $r_i$ . The goal of the online algorithm is to maximize the size of the matching.

The fractional version of the online unweighted matching is as follows: Each request  $r_i$  has a size  $x_i$  which is the amount of work needed to service request  $r_i$ . Algorithm  $A$  can match a fraction of size  $k_j^i \in [0, x_i]$  to each vertex  $s_j \in S$  adjacent to  $r_i$ . If request  $i$  is matched partially to some server  $j$  with weight  $k_j^i$  then  $\sum_{j=1}^m k_j^i \leq x_i$  and we have to maintain that the load of each server  $j$  denoted by  $l_j$ , which is  $\sum_{i=1}^n k_j^i$  where  $n$  is the length of  $\sigma$ , is at most 1. The

goal of the online algorithm is to maximize  $\sum_{j=1}^m l_j$ , where this is the *size of the fractional matching*  $M$ , denoted by  $|M|$ .

We show that a natural “water level” algorithm  $WL$  for the fractional matching problem is  $\frac{e}{e-1} \approx 1.58$ -competitive, even against a fractional  $OPT$ . Intuitively, given a request  $r_i \in R$ , algorithm  $WL$  flattens the load on the minimum loaded servers adjacent to  $r_i$ .

**Algorithm  $WL$ :** For each request  $r_i$ , match a fraction of size  $k_j^i$  for each adjacent  $s_j$ , where  $k_j^i = (h - l_j)_+$  and  $h \leq 1$  is the maximum number such that  $\sum_{j=1}^m k_j^i \leq x_i$ . By  $(f)_+$  we mean  $\max\{f, 0\}$ .

**Theorem 1.** *Algorithm  $WL$  is  $\frac{e}{e-1} \approx 1.58$ -competitive.*

**Theorem 2.** *Algorithm  $WL$  is  $\frac{e^\alpha}{e^\alpha-1}$ -competitive in a resource augmentation model, where the online algorithm can match  $\alpha$  times more than the optimum algorithm on a single server.*

*Remark 1.* Our technique can be extended for analyzing the b-matching problem that was studied in [9]. Specifically, we simplified the proof of the competitiveness of algorithm *Balance* for the b-matching problem which was studied by Kalyanasundaram and Pruhs [9].

## 4 The Maximizing Switch Throughput Problem - The Fractional Version

In this section we consider a fractional model, which is a relaxation of the discrete model which was presented in Section 2. In the fractional model we allow the online algorithm to accept *fractions* of packets into each queue if there is sufficient space, and also to transmit fractional of packets from the head of each queue s.t. the sum of transmitted fractions is at most 1.

We assume that sequence  $\sigma$  consists of *integral* packets. However, this restriction is not obligatory for this section. The restriction is relevant for the transformation of a fractional algorithm for the switch throughput problem into a discrete scheduling algorithm. We focus on algorithms that accept packets or fractional packets if sufficient place exists; since all packets have unit values there is no reason to prefer one packet over another. We refer to such algorithms as greedy admission control algorithms, and therefore focus on the scheduling problem alone. We show that a fractional algorithm for the scheduling problem is  $\frac{e}{e-1} \approx 1.58$ -competitive, even against a fractional adversary. We begin by introducing a translation of our problem (the fractional model) into the problem of online unweighted fractional matching in a bipartite graph.<sup>2</sup>

Given a sequence  $\sigma$ , we translate it into the bipartite graph  $G^\sigma = (R, S, E)$ , which is defined as follows:

- Let  $T$  denote the latest time unit in  $\sigma$  in which a packet arrives. We define the set of *time nodes* as  $R = \{r_1, \dots, r_{T+mB}\}$ . Each  $r_i \in R$  has unit size, i.e.  $x_i = 1$  for each  $1 \leq i \leq T + mB$

<sup>2</sup> A similar translation was presented in [4] for integral scheduling.

- Let  $P$  be the total number of packets specified in  $\sigma$ . We define the set of *packet nodes* as  $S = \{s_1, \dots, s_P\}$ .
- Let  $P_i^t$  denote the set of the last  $B$  packets that arrive at queue  $q_i$  until time  $t$  (inclusive). Define  $P^t = \bigcup_{i=1}^m P_i^t$ . We define the set of edges in  $G^\sigma$  as follows:  $E = \{(r_t, s_p) | p \in P^t\}$ .

Note that in  $G^\sigma$ , time and packets nodes arrive in an online fashion. Thus the problem is how to divide the time among the packets, where edges connect each of the time nodes to the packet nodes which correspond to packets that can be resident in some queue in time  $t$ . For consistency with Section 3 we denote the *time nodes* by **requests** and the *packet nodes* by **servers**.

*Remark 2.* We note that in contrast to the matching model that was presented in Section 3, here the servers as well as the requests arrive online. We emphasize that this does not change the results obtained in Section 3 since in  $G^\sigma$  the servers which arrive at time  $t$  only connect to request  $r_{t'} \geq t$  and thus can be viewed as servers which were present but not yet adjacent to requests  $r_{t'} < t$ .

Before we proceed we introduce some new definitions.

**Definition 1.** A schedule  $SC$  for a sequence of arriving packets  $\sigma$ , for the switch throughput problem, is a set of triplets of the form  $(t, q_i, k_i^t)$  for each  $1 \leq i \leq m$ , where queue  $q_i$  is scheduled for the transmission of a fraction of size  $k_i^t \geq 0$  at time  $t$ . The size of the schedule, denoted by  $|SC|$ , is the total amount of fractions scheduled for transmission, i.e.  $\sum_{t,i} k_i^t$ .

**Definition 2.** A schedule  $SC$  for a sequence  $\sigma$  is called *legal* if for every triplet  $(t, q_i, k_i^t)$ , queue  $q_i$  has a load of at least  $k_i^t$  at time  $t$  and  $\sum_{i=1}^m k_i^t \leq 1$ .

The following lemmas connect bipartite fractional matching to our problem.

**Lemma 1.** Every legal fractional schedule  $SC$  for the sequence  $\sigma$  can be mapped, in an online fashion, to a fractional matching  $M$  in  $G^\sigma$  such that  $|SC| = |M|$ .

**Lemma 2.** Every matching  $M$  in  $G^\sigma$  can be translated, in an online fashion in polynomial time, to a legal schedule  $SC$  for  $\sigma$  such that  $|SC| = |M|$ , while performing greedy admission control.

The following corollary is a directly result from Lemmas 1 and 2.

**Corollary 1.** For any sequence  $\sigma$ , the size of the optimal fractional schedule for  $\sigma$  is equal to the size of a maximum fractional matching in  $G^\sigma$ . In particular, optimal fractional schedule (which is equal to optimal integral matching) can be found (offline) in polynomial time.

Now, we present a fractional scheduling algorithm  $EP$  for maximizing switch throughput in the relaxed model. Since the bipartite unweighted fractional matching problem is connected to the maximizing switch throughput problem, algorithm  $EP$  intuitively bases its scheduling decisions on the matching constructed by algorithm  $WL$ , which was presented in Section 3, in the online constructed graph  $G^\sigma$ .

**Algorithm  $EP$ :**

- Maintain a running simulation of  $WL$  in the online constructed graph  $G^\sigma$ .
- **Admission control:** perform greedy admission control.
- **Scheduling:** transmit the total size of the fractions that were matched from  $r_t$  to servers in  $P_i^t$  from the head of queue  $i$ .

*Remark 3.* We note that algorithm  $EP$  actually constructs a legal schedule for  $\sigma$  incrementally as shown by Lemma 2.

**Theorem 3.** *For every sequence  $\sigma$ ,  $OPT(\sigma) \leq \frac{e}{e-1} EP(\sigma)$ .*

## 5 The Maximizing Switch Throughput Problem

In this section we consider the maximizing switch throughput problem. Given a sequence  $\sigma$  which consists of *integral* packets, we present a generic technique to transform any  $c$ -competitive online fractional algorithm for the switch throughput problem into a discrete algorithm with a competitive ratio of  $c(1 + \frac{\lfloor H_m \rfloor + 1}{B})$ , where  $H_m = \sum_{i=1}^m \frac{1}{i} \approx \ln m$  is the  $m$ th harmonic number. In particular, we take the fractional scheduling algorithm  $EP$ , which was presented in Section 4, with a competitive ratio of  $\frac{e}{e-1}$  and transform it into an online discrete scheduling algorithm with a competitive ratio of  $\frac{e}{e-1}(1 + \frac{\lfloor H_m \rfloor + 1}{B})$ . Thus, the competitive ratio of the discrete algorithm asymptotically approaches  $\frac{e}{e-1} \approx 1.58$  for large size queues  $B$ . We start by defining the cost scheduling problem in the next subsection.

### 5.1 The Cost Scheduling Problem

In this model queues are unbounded and packets **must** be inserted by the switching algorithm into each queue. The goal is to minimize the maximum cost, i.e. minimize the maximum queue size over time.

We now turn to consider a relaxation of the model and allow an online algorithm to transmit fractional packets waiting at the head of different non-empty queues, if they exist, provided that the total size of the transmitted fractions in one unit of time does not exceed 1. We denote by  $A$  the online fractional algorithm. We now introduce a general technique for the discretization of the scheduling of algorithm  $A$  given a finite sequence of *integral* packets  $\sigma$  while adding an additive factor of at most  $H_m$  to the cost of  $A$ . We start with the following definition and lemmas.

**Definition 3.** *An online fractional algorithm  $A$  is work-conserving if in each unit of time it transmits a total size of 1 if it exists, and transmits the total load if not.*

**Lemma 3.** *Every algorithm  $A$  can be transformed into a work-conserving algorithm  $A'$  while not worsening the performance of any sequence (in particular,  $c_{A'} \leq c_A$ , where  $c_{A'}$ ,  $c_A$  are the competitive ratios of  $A'$  and  $A$ , respectively).*

Henceforth we will deal exclusively with work-conserving algorithms, even when we neglect to say so explicitly. Now, we define algorithm  $M$  which gets an algorithm  $A$  as a parameter and denote it by  $M^A$ . At each time unit, in order to decide which queue to serve,  $M^A$  computes the load seen by  $A$  and uses this information to make its decision. Given a time unit  $t$ , let  $l_i^A$  and  $l_i$  be the simulated load of  $A$  and the actual load of  $M^A$  on queue  $i$  at that time, respectively. The *residual load* of queue  $i$  at time unit  $t$  is defined as  $l_i^{res} = l_i - l_i^A$ . Intuitively, algorithm  $M^A$  transmits at each time unit from a queue with maximum residual load. Before we present the exact definition of algorithm  $M^A$  we define the following:

**Definition 4.** *The mid-state of each time unit is the state of the queues after algorithm  $A$  transmits its fractional packets from some queues, and just before algorithm  $M^A$  transmits its packet from some queue.*

**Algorithm  $M^A$ :** Maintain a running simulation of algorithm  $A$ . At each time unit transmit a packet from a queue which has the maximum residual load at the mid-state (breaking ties arbitrarily), unless all queues are empty.

**Theorem 4.** *The cost of  $M^A$  is at most the cost of  $A$  plus  $H_m$ .*

*Remark 4.* An alternative proof of a slightly weaker bound of  $\lfloor \log_2 m \rfloor$  plus the cost of  $A$  can be obtained using [6], non explicitly.

## 5.2 Discretization of the Fractional Scheduling

Given a finite sequence of *integral* packets  $\sigma$  we present a general technique to transform any  $c$ -competitive *fractional* algorithm for the switch throughput problem into a discrete algorithm with a competitive ratio of  $c(1 + \frac{\lfloor H_m \rfloor + 1}{B})$ . In particular, we will apply this technique for the discretization of algorithm  $EP$ , which was presented in Section 4. It can easily be shown that any algorithm  $A$  can be transformed into a greedy admission control algorithm  $A'$  while not worsening the performance of any sequence (in particular,  $c_{A'} \leq c_A$ , where  $c_{A'}$ ,  $c_A$  are the competitive ratios of  $A'$  and  $A$ , respectively); we focus on greedy admission control algorithms.

For our discretization process we rely on the results of the cost problem, which were presented in Subsection 5.1. We want to address the packets which were accepted by  $EP$  as the input sequence  $\sigma$  for the cost problem studied in Subsection 5.1, in a manner which is yet to be seen. Recall that algorithm  $EP$  is a greedy admission control algorithm. Thus,  $EP$  might accept a fractional packets due to insufficient queue space. Since in the model of the cost problem we study the case where  $\sigma$  consists of integral packets we want  $EP$  to only accept packets integrally. Hence, we continue by considering the following problem: assume we are given an online  $c$ -competitive algorithm  $A$  with greedy admission control. We want to produce a competitive algorithm  $\hat{A}$  which assigns only integral packets. We start by assuming that algorithm  $\hat{A}$  has queues of size  $B + 1$  (algorithm  $A$  maintains queues of size  $B$ ); we shall get rid of this assumption later on.

Intuitively,  $\hat{A}$  emulates the scheduling of  $A$  and accepts only integral packets. We start with a definition and an observation before we define  $\hat{A}$  formally. S

**Definition 5.** *We denote algorithms that accept integral packets if there is sufficient space, as discrete greedy admission control algorithms.*

**Observation 1.** *Every greedy admission control algorithm assigns fractions on a queue only when the load on that queue is strictly above  $B - 1$ .*

We now define the transformation of a given greedy admission control algorithm  $A$  with queues of size  $B$  into algorithm  $\hat{A}$  with queues of size  $B + 1$  which only accepts integral packets.

**Algorithm  $\hat{A}$ :**

- Maintain a running simulation of  $A$ . Let  $k_i^t$  be the fraction size which was transmitted by algorithm  $A$  at time  $t$  from queue  $i$ .
- **Admission control:** Perform discrete greedy admission control.
- **Scheduling:** For each queue  $i$  transmit a fraction of size  $k_i^t$ .

Let  $l_i^t$  and  $\hat{l}_i^t$  be the loads of queue  $i$  in a given time unit  $t$  in  $A$  and  $\hat{A}$ , respectively. For the algorithm to be well defined, i.e.  $\hat{l}_i^t \geq k_i^t$  after the arrival phase, we must prove the next lemma.

**Lemma 4.** *For each queue  $i$  and time unit  $t$ , after the arrival phase  $\hat{l}_i^t \geq l_i^t$ .*

**Corollary 2.** *Algorithm  $\hat{A}$  can always transmit  $k_i^t$  as defined.*

**Theorem 5.** *For a given algorithm  $A$  with queues of size  $B$ , algorithm  $\hat{A}$  with queues of size  $B + 1$  has the same throughput given the same sequence  $\sigma$ .*

*Proof.* From Corollary 2, at each time unit  $t$ ,  $\hat{A}$  transmits the same total size as algorithm  $A$ .

Recall that  $EP$  is not a work-conserving algorithm. Therefore  $\hat{EP}$  which emulates the scheduling of  $EP$  is also not a work-conserving algorithm. Since we want to use some of the results from the cost problem in Subsection 5.1, we first need to transform algorithm  $\hat{EP}$  into a work-conserving algorithm  $\hat{EP}'$ . We use the transformation presented by Lemma 3 in Subsection 5.1.

**Lemma 5.** *Every non work-conserving algorithm  $A$  can be transformed into a work-conserving algorithm  $A'$  which accepts only packets which are accepted by  $A$ , while not worsening the performance of any sequence (in particular,  $c_{A'} \leq c_A$ , where  $c_{A'}$ ,  $c_A$  are the competitive ratios of  $A'$  and  $A$ , respectively).*

Now, we consider Algorithm  $M$  which was presented in Section 5.1. We assume that  $M$  has queues of size  $B + 1 + \lfloor H_m \rfloor$ . Recall that the precondition of  $M$  was that the parameter algorithm is work-conserving. Hence, we apply algorithm  $M$  on algorithm  $\hat{EP}'$  and denote the resulting algorithm as  $M^{\hat{EP}'}$ . The sequence of packets which is given to  $M^{\hat{EP}'}$  will consist **solely** of the packets which were accepted by  $\hat{EP}'$ . Note that this is a sequence which consists exclusively of integral packets. We emphasize that  $M^{\hat{EP}'}$  is a *discrete* algorithm, since it only transmits integral packets. We now present the following lemma.



**Lemma 6.** *Algorithm  $M^{\hat{E}P'}$  with queues of size  $B + 1 + \lfloor H_m \rfloor$  has the same throughput as algorithm  $\hat{E}P'$  with queues of size  $B + 1$ , given the same sequence.*

We now return to our original model where queues are of size  $B$ . By Lemma 6, if  $M^{\hat{E}P'}$  had queues of size  $B + \lfloor H_m \rfloor + 1$ , then algorithms  $M^{\hat{E}P'}$  and algorithm  $\hat{E}P'$  would have had equal throughput. Unfortunately, this is not the case, so we continue by emulating an algorithm with large queues with an algorithm of small queues. Specifically, assume we are given an online competitive discrete algorithm  $A$  with queues of size  $y$  and we want to produce a competitive algorithm  $E^A$  with queues  $y' < y$ . Intuitively,  $E^A$  tries to emulate the schedule of algorithm  $A$  and accept only packets which  $A$  accepts as long as the queue is not full. We emphasize that if algorithm  $A$  was a fractional algorithm this problem could easily be solved by scaling the accepted volume and the transmitted volume of  $A$  by  $y'/y$ . Thus we continue by considering only discrete algorithms.

**Algorithm  $E^A$ :**

- Maintain a running simulation of algorithm  $A$ .
- **Admission control:** accept a packet if  $A$  accepts it and the queue is not full.
- **Scheduling:** transmit packets as  $A$  if the queue is not empty.

We show that the competitiveness of  $E^A$  is the competitive ratio of  $A$  times the ratio  $y/y' > 1$ .

**Theorem 6.** *Let  $A$  maintain queues of size  $y$  and let  $E^A$  maintain queues of size  $y'$  such that  $y' < y$ . Then  $A(\sigma) \leq \frac{y}{y'} E^A(\sigma)$ .*

We prove the main result of this paper with the next theorem.

**Theorem 7.** *The competitive ratio of algorithm  $E^{M^{\hat{E}P'}}$  for the maximizing switch throughput problem is  $\frac{e}{e-1}(1 + \frac{\lfloor H_m \rfloor + 1}{B})$ .*

**Corollary 3.** *For  $B \gg H_m$  the competitive ratio of  $E^{M^{\hat{E}P'}}$  approaches  $\frac{e}{e-1} \approx 1.58$ .*

## 6 Discussion and Open Problems

- Albers and Schmidt [2] show a lower bound of  $\frac{e}{e-1}$ , for  $m \gg B$ . They also show an algorithm with competitiveness  $\frac{17}{9} \approx 1.89$  for every  $B > 1$ . Our main result in this paper shows that for the case were  $B > \log m$  our algorithm can do much better than 1.89. Our algorithm approaches a competitive ratio of  $\frac{e}{e-1}$ , which interestingly is the lower bound for  $m \gg B$ . Azar and Richter [4] show a lower bound of  $1.366 - \Theta(\frac{1}{m})$  for any  $B$  and  $m$ , which is smaller than the lower bound of  $\frac{e}{e-1}$ . Therefore, there is still work to be done to determine the optimal competitive ratio for arbitrary  $B$  and  $m$ .
- It is an interesting question whether greedy algorithms which transmits from the longest queue first has a competitive ratio smaller than 2, for  $B \gg m$ .



- For  $B \gg \log m$  our general technique shows that the competitive ratio of the discrete version algorithm approaches the performance of a fractional algorithm. Hence, it is interesting to determine the best fractional algorithm for large sized queues.

## References

1. W. A. Aiello, Y. Mansour, S. Rajagopalan, and A. Rosen. Competitive queue policies for differentiated services. In *Proceedings of the IEEE INFOCOM '2000*, pages 431–440, 2000.
2. S. Albers and M. Schmidt. On the performance of greedy algorithms in packet buffering. In *Proc. 36th ACM Symp. on Theory of Computing*, 2004.
3. N. Andelman, Y. Mansour, and A. Zhu. Competitive queueing policies for QoS switches. In *Proc. 13rd ACM-SIAM Symp. on Discrete Algorithms*, pages 761–770, 2003.
4. Y. Azar and Y. Richter. Management of multi-queue switches in QoS networks. In *Proc. 35th ACM Symp. on Theory of Computing*, pages 82–89, 2003.
5. Y. Azar and Y. Richter. The zero-one principle for switching networks. In *Proc. 36th ACM Symp. on Theory of Computing*, 2004.
6. A. Bar-Noy, A. Freund, S. Landa, and J. Naor. Competitive on-line switching policies. In *Proc. 13rd ACM-SIAM Symp. on Discrete Algorithms*, pages 525–534, 2002.
7. A. Birman, H. R. Gail, S. L. Hantler, Z. Rosberg, and M. Sidi. An optimal service policy for buffer systems. *Journal of the Association Computing Machinery (JACM)*, 42(3):641–657, 1995.
8. M. Chrobak, J. Csirik, C. Imreh, J. Noga, J. Sgall, and G. J. Woeginger. The buffer minimization problem for multiprocessor scheduling with conflicts. In *Proc. 28th International Colloquium on Automata, Languages, and Programming*, pages 862–874, 2001.
9. B. Kalyanasundaram and K. R. Pruhs. An optimal deterministic algorithm for online b-matching. *Theoretical Computer Science*, 233:319–325, 2000.
10. R. Karp, U. Vazirani, and V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of 22nd Annual ACM Symposium on Theory of Computing*, pages 352–358, may 1990.
11. A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in QoS switches. In *Proc. 33rd ACM Symp. on Theory of Computing*, pages 520–529, 2001.
12. A. Kesselman and Y. Mansour. Loss-bounded analysis for differentiated services. In *Proc. 12nd ACM-SIAM Symp. on Discrete Algorithms*, pages 591–600, 2001.
13. H. Koga. Balanced scheduling toward loss-free packet queuing and delay fairness. In *Proc. 12th Annual International Symposium on Algorithms and Computation*, pages 61–73, 2001.
14. Z. Lotker and B. Patt-Shamir. Nearly optimal fifo buffer management for diffserv. In *Proc. 21st ACM Symp. on Principles of Distrib. Computing*, pages 134–143, 2002.
15. M. May, J. C. Bolot, A. Jean-Marie, and C. Diot. Simple performance models of differentiated services for the internet. In *Proceedings of the IEEE INFOCOM '1999*, pages 1385–1394, 1999.

# An Improved Algorithm for CIOQ Switches

Yossi Azar\* and Yossi Richter\*\*

School of Computer Science, Tel Aviv University, Tel Aviv, 69978, Israel.  
{azar,yo}@tau.ac.il

**Abstract.** The problem of maximizing the weighted throughput in various switching settings has been intensively studied recently through competitive analysis. To date, the most general model that has been investigated is the standard CIOQ (Combined Input and Output Queued) switch architecture with internal fabric speedup  $S \geq 1$ . CIOQ switches, that comprise the backbone of packet routing networks, are  $N \times N$  switches controlled by a switching policy that incorporates two components: admission control and scheduling. An admission control strategy is essential to determine the packets stored in the FIFO queues in input and output ports, while the scheduling policy conducts the transfer of packets through the internal fabric, from input ports to output ports. The online problem of maximizing the total weighted throughput of CIOQ switches was recently investigated by Kesselman and Rosén in [12]. They presented two different online algorithms for the general problem that achieve non-constant competitive ratios (linear in either the speedup or the number of distinct values or logarithmic in the value range). We introduce the first constant-competitive algorithm for the general case of the problem, with arbitrary speedup and packet values. Specifically, our algorithm is 9.47-competitive, and is also simple and easy to implement.

## 1 Introduction

*Overview:* Recently, packet routing networks have become the dominant platform for data transfer. The backbone of such networks is composed of  $N \times N$  switches, that accept packets through multiple incoming connections and route them through multiple outgoing connections. As network traffic continuously increases and traffic patterns constantly change, switches routinely have to efficiently cope with overloaded traffic, and are forced to discard packets due to insufficient buffer space, while attempting to forward the more valuable packets to their destinations.

Traditionally, the performance of queuing systems has been studied within the stability analysis framework, either by a probabilistic model for packet injection (queuing theory, see e.g. [7,14]) or an adversarial model (adversarial queuing theory, see e.g. [4,8]). In stability analysis packets are assumed to be identical,

---

\* Research supported in part by the Israeli Ministry of industry and trade and by the Israel Science Foundation.

\*\* Research supported in part by the Israeli Ministry of industry and trade.

and the goal is to determine queue sizes such that no packet is ever dropped. However, real-world networks do not usually conform with the above assumptions, and it seems inevitable to drop packets in order to maintain efficiency. As a result, the competitive analysis framework, which avoids any assumptions on the input sequence and compares the performance of online algorithms to the optimal solution, has been adopted recently for studying throughput maximization problems. Initially, online algorithms for single-queue switches were studied in various settings [1,3,10,11,13]. Later on, switches with multiple input queues were investigated [2,5,6], as well as CIOQ switches with multiple input and output queues [12].

To date, the most general switching model that has been studied using competitive analysis is CIOQ (Combined Input and Output Queued) switching architecture. A CIOQ switch with speedup  $S \geq 1$  is an  $N \times N$  switch, with  $N$  input ports and  $N$  output ports. The internal fabric that connects the input and output FIFO queues is  $S$  times faster than the queues. A switching policy for a CIOQ switch consists of two components. First, an admission control policy to determine the packets stored in the bounded-capacity queues. Second, a scheduling strategy to decide which packets are transferred from input ports to output ports through the intermediate fabric at each time step. The goal is to maximize the total value of packets transmitted from the switch. The online problem of maximizing the total throughput of a CIOQ switch was studied by Kesselman and Rosén in [12]. For the special case of unit-value packets (all packets have the same value) they presented a greedy algorithm that is 2-competitive for a speedup of 1 and 3-competitive for any speedup. For the general case of packets with arbitrary values two different algorithms were presented, by using techniques similar to those introduced in [5]. The first algorithm is  $4S$ -competitive and the second one is  $(8 \min\{k, 2\lceil \log \alpha \rceil\})$ -competitive, where  $k$  is the number of distinct packet values and  $\alpha$  is the ratio between the largest and smallest values.

*Our results:* We present the first constant-competitive algorithm for the general case of the problem with arbitrary packet values and any speedup. Specifically, our algorithm is 9.47-competitive and is also simple and easy to implement. Our analysis includes a new method to map discarded packets to transmitted packets with the aid of generating dummy packets. We note that our algorithm and its analysis apply almost as-is to less restrictive (and less common) models of CIOQ switches, such as switches with priority queues rather than FIFO queues.

*Other related results:* The online problem of throughput maximization in switches has been explored extensively during recent years. Initially, single-queue switches were investigated, for both arbitrary packet sequences, and 2-value sequences. The preemptive model, in which packets stored in the queue can be discarded, was studied in [10,11,13]. The non-preemptive model, in which a packet can be discarded only upon arrival, was initially studied by Aiello *et al.* [1], followed by Andelman *et al.* [3] who showed tight bounds. These results for a single queue were generalized for switches with arbitrary number of input queues in [5], by a general reduction from the multi-queue model to the single-queue

model. Specifically, a 4-competitive algorithm is presented in [5] for the weighted multi-queue switch problem. An improved 3-competitive algorithm was recently shown in [6]. The multi-queue switch model has been also investigated for the special case of unit-value packets, which corresponds to IP networks. Albers and Schmidt [2] recently introduced a deterministic 1.89-competitive algorithm for the unit-value problem; A randomized 1.58-competitive algorithm was previously shown in [5]. An alternative model to the multiple input queues model is the shared memory switch, in which memory is shared among all queues. Hahne *et al.* [9] studied buffer management policies in this model while focusing on deriving upper and lower bounds for the natural Longest Queue Drop policy.

## 2 Problem Definition and Notations

In the online CIOQ (Combined Input and Output Queued) switch problem, originally introduced in [12], we are given an  $N \times N$  switch with  $N$  input ports and  $N$  output ports (see figure 1). Input port  $i$  ( $i = 1, \dots, N$ ) contains  $N$  virtual output FIFO queues (referred to as *input queues* henceforth), denoted  $\{VO_{ij}\}_{j=1}^N$ , with bounded capacities, where queue  $VO_{ij}$  is used to buffer the packets arriving at input port  $i$  and destined to output port  $j$ . Output port  $j$  ( $j = 1, \dots, N$ ) contains a bounded capacity FIFO queue (also referred to as *output queue*), denoted  $O_j$ , to buffer the packets waiting to be transmitted through output port  $j$ . For each queue  $Q$  in the system, we denote by  $B(Q)$  the capacity of the queue, and by  $Q(t)$  the ordered set of packets stored in the queue at time step  $t$ . We denote by  $h(Q(t))$  and  $\min(Q(t))$  the packet at the head of queue  $Q$  and the minimal value, respectively.

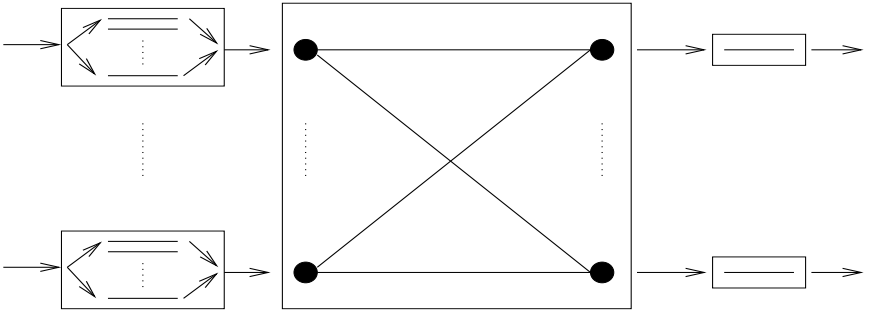


Fig. 1. CIOQ switch - an example

Time proceeds in discrete time steps. We divide each time step into three phases. The first phase is the *arrival phase*, during which packets arrive online at the input ports. All packets have equal size, and each packet is labelled with a value, an input port, through which it enters the switch, and a destination output port, through which it has to leave the switch. We denote the finite sequence of

online arriving packets by  $\sigma$ . For every packet  $p \in \sigma$  we denote by  $v(p)$ ,  $in(p)$  and  $out(p)$  its value, input port and output port, respectively. Online arriving packets can be enqueued in the input queue that corresponds to their destination port, without exceeding its capacity. Remaining packets must be discarded. We allow preemption, i.e. packets previously stored in the queues may be discarded in order to free space for newly arrived packets. The second phase at each time step is the *scheduling phase* during which packets are transferred from the input queues to the output queues. For a switch with speedup  $S \geq 1$  at most  $S$  packets can be removed from each input port and at most  $S$  packets can be enqueued into the queue of each output port. This is done in  $S$  consecutive rounds, during each we compute a matching between input and output ports. We denote round  $s$  ( $s = 1, \dots, S$ ) at time step  $t$  by  $t_s$ . During the scheduling phase the capacity of the output queues may not be exceeded, and again preemption is allowed. The third phase at each time step is the *transmission phase*. During this phase a packet may be transmitted from the head of each output queue.

A switching algorithm  $\mathcal{A}$  for the CIOQ switch problem is composed of two components, not necessarily independent. The first component is admission control; Algorithm  $\mathcal{A}$  has to exercise an admission control strategy in all queues (input and output), that decides which packets are admitted into the queues and which packets are discarded in case of overflow. The second component is a scheduling strategy; During scheduling round  $t_s$  algorithm  $\mathcal{A}$  first decides the set of eligible packets for transfer  $E^{\mathcal{A}}(t_s) \subseteq \{h(VO_{ij}(t_s)) : (i, j) \in [N]^2\}$ , i.e. a subset of the packets currently located at the head of the input queues. The set  $E^{\mathcal{A}}(t_s)$  can be translated into the bipartite weighted graph  $G^{\mathcal{A}}(t_s) = ([N], [N], E)$  such that  $E = \{(in(p), out(p)) : p \in E^{\mathcal{A}}(t_s)\}$ , where the weights on the edges correspond to the packet values, namely  $w(e) = v(p)$  for  $e = (in(p), out(p)) \in E$ . Algorithm  $\mathcal{A}$  then constructs a matching  $M^{\mathcal{A}}(t_s) \subseteq G^{\mathcal{A}}(t_s)$  that corresponds of the set of packets that are transferred from input queues to output queues at round  $t_s$ . Given a finite sequence  $\sigma$  we denote by  $\mathcal{A}(\sigma)$  the set of packets algorithm  $\mathcal{A}$  transmitted from the switch. The goal is to maximize the total value of transmitted packets, denoted  $V(\mathcal{A}(\sigma))$ , i.e. maximize  $V(\mathcal{A}(\sigma)) = \sum_{p \in \mathcal{A}(\sigma)} v(p)$ . An online algorithm  $\mathcal{A}$  is said to be  $c$ -competitive if and only if for every packet sequence  $\sigma$ ,  $V(\text{Opt}(\sigma)) \leq c \cdot V(\mathcal{A}(\sigma))$  holds, where  $\text{Opt}$  denotes the optimal off-line algorithm for the problem.

### 3 The Algorithm

In this section we define and analyze our algorithm for the CIOQ switch problem. We begin with a definition of a parameterized preemptive admission control policy  $\text{GR}_\beta$  for a single queue (figure 2). This policy is a generalization of the ordinary greedy policy from [10], that is obtained by setting  $\beta = 1$ . The latter will be denoted by  $\text{GR}$ . We then present our parameterized algorithm  $\text{SG}(\beta)$  (abbreviation for **Semi-Greedy**( $\beta$ )) for the problem (figure 3). For simplicity of notation, we drop the parameter  $\beta$  for the remainder of this section, and use

**Algorithm  $\text{GR}_\beta$  [Single-Queue]**

Enqueue a new packet  $p$  if:

- $|Q(t)| < B(Q)$  (the queue is not full).
- Or  $v(p) > \beta \cdot \min(Q(t))$ . In this case the smallest packet is discarded and  $p$  is enqueued.

**Fig. 2.** Algorithm  $\text{GR}_\beta$ .**Algorithm  $\text{SG}(\beta)$  [CIOQ switch]**

1. **Admission control:**
  - a) Input queues: use algorithm GR.
  - b) Output queues: use algorithm  $\text{GR}_\beta$ .
2. **Scheduling:** at scheduling round  $t_s$  do:
  - a) Set  $E^{\text{SG}}(t_s) = \{h(VO_{ij}(t_s)) : |O_j(t_s)| < B(O_j) \text{ or } v(h(VO_{ij}(t_s))) > \beta \cdot \min(O_j(t_s))\}$
  - b) Compute a maximum weighted matching  $M^{\text{SG}}(t_s) \subseteq G^{\text{SG}}(t_s)$ .

**Fig. 3.** Algorithm  $\text{SG}(\beta)$ .

the abbreviated notation  $\text{SG}$  for the algorithm. The value of the parameter  $\beta$  will be determined later.

Algorithm  $\text{SG}$  exercises the greedy preemptive admission control policy GR on all input queues, and the modified admission control policy  $\text{GR}_\beta$  on all output queues. At each scheduling round  $\text{SG}$  considers only those packets that would be admitted to their destined output queue if transferred, namely packets whose destined output queue is either not full, or they have a value greater than  $\beta$  times the currently smallest packet in this queue. Algorithm  $\text{SG}$  then computes a maximum weighted matching in the corresponding bipartite graph. Note that according to  $\text{SG}$  operation, packets can be discarded from output queues during each scheduling round. We now proceed to prove the competitive ratio of  $\text{SG}$ .

**Theorem 1.** *Algorithm  $\text{SG}$  achieves constant competitive ratio. Specifically, for an optimal choice of the parameter  $\beta$ , algorithm  $\text{SG}$  is 9.47-competitive.*

*Proof.* The outline of the proof is as follows. Given an input sequence  $\sigma$ , we wish to construct a global weight function  $w_\sigma : \text{SG}(\sigma) \rightarrow \mathbb{R}$ , that maps each packet transmitted by  $\text{SG}$  from the switch to a real value. In order to achieve a constant competitive ratio it is sufficient to prove the following:

1.  $\sum_{p \in \text{Opt}(\sigma) \setminus \text{SG}(\sigma)} v(p) \leq \sum_{p \in \text{SG}(\sigma)} w_\sigma(p)$ .
2.  $w_\sigma(p) \leq c \cdot v(p)$ , for each packet  $p \in \text{SG}(\sigma)$ , and for some global constant  $c$ .

Note that by the first condition the total weight that is mapped to packets which were transmitted by  $\text{SG}$  covers the total value lost by  $\text{SG}$  compared with  $\text{Opt}$ .

By the second condition this is done by charging each packet with a weight that exceeds its own by no more than a constant factor. In the following we show how to construct  $w_\sigma$ .

*Step 1: The basic mapping scheme.* We begin by introducing some additional notations. Consider the single-queue admission control policy  $\text{GR}_\beta$ , and let  $\mathcal{A}$  denote any single-queue admission control policy that is exercised by  $\text{Opt}$ . Assume that  $\text{GR}_\beta$  and  $\mathcal{A}$  do not operate under the same conditions, specifically, let  $\sigma_1$  and  $\sigma_2$  be the input packet sequences given to  $\mathcal{A}$  and  $\text{GR}_\beta$ , and let  $T_1$  and  $T_2$  denote the time steps at which  $\mathcal{A}$  and  $\text{GR}_\beta$ , respectively, transmit from the queue. We denote by  $\sigma^{\mathcal{A}-\text{GR}_\beta} = \{p \in \sigma_1 \cap \sigma_2 : p \in \mathcal{A}(\sigma_1) \setminus \text{GR}_\beta(\sigma_2)\}$  the set of packets in the intersection of the sequences that were transmitted by  $\mathcal{A}$  and not by  $\text{GR}_\beta$ . The following online-constructed local mapping will serve as the basic building block in the construction of our global weight function  $w_\sigma$ .

**Lemma 1.** *Let  $\mathcal{A}$  and  $\text{GR}_\beta$  (for  $\beta \geq 1$ ) be admission control policies for a single queue, with input packet sequences  $\sigma_1, \sigma_2$  and transmission times  $T_1, T_2$  respectively. If  $T_1 \subseteq T_2$  then there exists a mapping  $\mathbf{M}_\mathbf{L} : \sigma^{\mathcal{A}-\text{GR}_\beta} \rightarrow \text{GR}_\beta(\sigma_2)$  such that the following properties hold:*

1.  $v(p) \leq \beta \cdot v(\mathbf{M}_\mathbf{L}(p))$ , for every packet  $p \in \sigma^{\mathcal{A}-\text{GR}_\beta}$ .
2. Every packet in  $\text{GR}_\beta(\sigma_2) \setminus \mathcal{A}(\sigma_1)$  is mapped to at most twice.
3. Every packet in  $\text{GR}_\beta(\sigma_2) \cap \mathcal{A}(\sigma_1)$  is mapped to at most once.

*Proof.* For simplicity of notation, we prove the lemma for  $\text{GR}$ , i.e.  $\text{GR}_1$ . The lemma will then follow directly for the general case, using the same construction, since for  $\text{GR}_\beta$  the ratio between a discarded packet and any packet in the queue is at most  $\beta$ . We construct the mapping  $\mathbf{M}_\mathbf{L}$  on-line following the operation of the admission control policies. For the remainder of the proof we denote the contents of the queue according to  $\text{GR}$  operation (respectively  $\mathcal{A}$  operation) by  $Q_{\text{GR}}$  (respectively by  $Q_\mathcal{A}$ ). We further denote by  $\mathbf{M}_\mathbf{L}^{-1}(p)$  the packets that are mapped to packet  $p$  (note that  $|\mathbf{M}_\mathbf{L}^{-1}(p)| \leq 2$  for every packet  $p$ ). Given a packet  $p \in Q_{\text{GR}}(t)$  we denote by  $\text{potential}(p)$  the maximum number of packets that can be mapped to  $p$  according to the properties of the lemma, namely  $\text{potential}(p) = 1$  for  $p \in \mathcal{A}(\sigma_1)$ , otherwise  $\text{potential}(p) = 2$ . We further denote by  $\text{available}_t(p)$  the number of available mappings to  $p$  at time  $t$ , namely  $\text{potential}(p)$  minus the number of packets already mapped to  $p$  until time  $t$ . A packet  $p \in Q_{\text{GR}}(t)$  is called *fully-mapped* if no additional packet can be mapped to it, i.e. if  $\text{available}_t(p) = 0$ . The definition of the mapping  $\mathbf{M}_\mathbf{L}$  appears in figure 4.

We begin by observing that the mapping  $\mathbf{M}_\mathbf{L}$  indeed adheres to the required properties of the lemma.

*Claim.* The mapping  $\mathbf{M}_\mathbf{L}$  meets properties 1–3 of Lemma 1

*Proof.* By induction on the time steps. First, observe that a packet is mapped to only if it is not fully-mapped, therefore properties 2–3 are satisfied. Second, by the definition of  $\text{GR}$ , when a packet  $p$  is discarded from the queue all other

**Mapping  $M_L : \sigma^{\mathcal{A}-GR} \rightarrow GR(\sigma_2)$**

1. Let  $p \in \sigma^{\mathcal{A}-GR}$  arrive at time  $t$ ; Map  $p$  to the packet closest to the head in queue  $Q_{GR}$  that is not fully-mapped.
2. Let  $q \in Q_{GR}(t)$  be a packet that is discarded from queue  $Q_{GR}$  during time  $t$ . Update  $M_L$  as follows:
  - a) If  $q \in \mathcal{A}(\sigma_1)$  map it to the packet closest to the head that is not fully-mapped.
  - b) Do the same with respect to  $M_L^{-1}(q)$  (if exists).

**Fig. 4.** Local mapping  $M_L$ .

packets residing in the queue have higher values. By the induction hypothesis, this is also true with respect to  $M_L^{-1}(p)$ . Therefore, all mappings concur with property 1. This completes the proof.  $\square$

We now turn to characterize the packets in queue  $Q_{GR}$  whenever a mapped packet resides in the queue.

*Claim.* Let  $p \in Q_{GR}(t)$  be a mapped packet in queue  $Q_{GR}$  at time step  $t$ . Then the following holds:

1.  $h(Q_{GR}(t))$  (the packet at the head) is mapped.
2. For every packet  $q \in M_L^{-1}(p)$  we have:  $v(q) \leq v(h(Q_{GR}(t)))$

*Proof.* If packet  $p$  is at the head of queue  $Q_{GR}$  at time  $t$  then we are clearly done by Claim 3. Otherwise, let  $t' \leq t$  be the time step at which packet  $p$  was mapped to for the last time. By the definition of  $M_L$  all the packets closer to the head than  $p$  were already fully-mapped at time  $t'$ . Furthermore, all those packets have higher values than the value of the packet (or packets) mapped to  $p$  at that time, according to the definition of  $GR$ . Since FIFO order is obtained, these properties continue to hold at time  $t$ , and the claim follows.  $\square$

To complete the proof of Lemma 1 we need to prove that the definition of  $M_L$  is valid, i.e. that indeed whenever a packet is discarded, queue  $Q_{GR}$  contains packets that are not fully-mapped.

*Claim.* The definition of the mapping  $M_L$  is valid, namely, at each time step the queue  $Q_{GR}$  contains unmapped packets as assumed in the definition of  $M_L$ .

*Proof.* We define a potential function  $\Phi(t) := \sum_{p \in Q_{GR}(t)} available_t(p)$ , that counts the number of available mappings at every time step. We prove the claim by induction on the changes that occur in the system, i.e. packet arrivals and transmissions. Specifically, the correctness of the claim follows from the invariant inequality  $\Phi(t) + |Q_{\mathcal{A}}(t)| \geq |Q_{GR}(t)|$  that is shown to hold at each time step by induction. For the initial state the inequality clearly holds. We next denote by  $\Delta_\Phi$ ,  $\Delta_{\mathcal{A}}$  and  $\Delta_{GR}$  the changes that occur in the values of  $\Phi$ ,  $|Q_{\mathcal{A}}(t)|$ ,  $|Q_{GR}(t)|$ , respectively. We examine all possible changes in the system, and show that the inequality continues to hold, either by proving it directly or by showing that



$\Delta_\Phi + \Delta_A \geq \Delta_{GR}$ . For the remainder of the proof we assume w.l.o.g that  $\mathcal{A}$  does not preempt packets.

1. **Packet  $p \in \sigma_1 \cap \sigma_2$  arrives:** We examine the three possible cases:
  - a)  **$p$  is accepted by GR while no packet is discarded:** Clearly,  $\Delta_\Phi \geq \Delta_{GR}$ .
  - b)  **$p$  is rejected by GR:** If  $p$  is also rejected by  $\mathcal{A}$  nothing changes. Otherwise, queue  $Q_A$  was not full before the arrival of  $p$  as opposed to queue  $Q_{GR}$ . Hence by the induction hypothesis before the packet arrival  $\Phi > 0$ . After we map a packet to  $p$ ,  $\Delta_\Phi + \Delta_A = 0$  and the inequality holds.
  - c)  **$p$  is accepted by GR while a packet is discarded:** Clearly,  $\Delta_{GR} = 0$  since a packet is discarded only when queue  $Q_{GR}$  is full. Note that in either case ( $p$  is accepted or rejected by  $\mathcal{A}$ )  $\Delta_\Phi + \Delta_A \geq 0$  since the left-hand side of the inequality first increases by two and then decreases by at most two (case 2 in  $M_L$  definition). Therefore, the inequality continues to hold.
2. **Packet  $p \in \sigma_2 \setminus \sigma_1$  arrives:**  $\Delta_\Phi \geq \Delta_{GR}$  whether  $p$  is accepted by GR or not.
3. **Packet  $p \in \sigma_1 \setminus \sigma_2$  arrives:**  $\Delta_A \geq 0$ , while other values remain unchanged.
4. **Transmission step  $t \in T_1 \cap T_2$ :** If queue  $Q_{GR}$  does not hold mapped packets, the inequality trivially holds, since in this case  $\Phi(t) \geq |Q_{GR}(t)|$ . Otherwise, by Claim 3 the packet at the head of the queue is a mapped packet. If the packet at the head of the queue is a fully-mapped packet then after the transmission takes place  $\Delta_A = \Delta_{GR} = -1$  while  $\Delta_\Phi = 0$  and the inequality holds; Otherwise, by the definition of  $M_L$ , there are no additional mapped packets in queue  $Q_{GR}$  and we are back to the previous case.
5. **Transmission step  $t \in T_2 \setminus T_1$ :** If the packet at the head of queue  $Q_{GR}$  is fully-mapped then after the transmission takes place  $\Delta_{GR} = -1$  while  $\Delta_\Phi = \Delta_A = 0$  and the inequality holds. Otherwise, after the transmission there are no mapped packets in the queue and, again, by the same considerations given in the previous case, the inequality goes on.

□

This completes the proof of Lemma 1.

□

*Step 2: Defining the global weight function - phase 1.* Note that algorithm SG can lose packets by one of two ways. Packets can be discarded at the input queues, and packets stored in the output queues can be preempted in favor of higher value packets. Ideally, we would like to proceed as follows. Apply the local mapping  $M_L$  to each input queue to account for packets discarded there. Then, whenever a packet  $p$  preempts a packet  $q$  at an output queue, map  $q$  along with the packets mapped to it to packet  $p$ . Now define the global weight function such that  $w_\sigma(p)$  equals the total value of packets mapped to  $p$ . However, the required condition for the construction of the mapping  $M_L$  (see Lemma 1) is not met; We have no guarantee that  $T_1 \subseteq T_2$ , namely that SG transmits from each input queue whenever Opt does. As we shall see this obstacle can be overcome with some extra cost.

**Creating dummy packets**

1. For scheduling round  $t_s$  define:

$$\begin{aligned} S(t_s) &:= \{(i, j) : VO_{ij} \text{ contains a mapped packet}\} \\ S_1(t_s) &:= \{(i, j) \in S(t_s) : (i, j) \in M^{\text{Opt}}(t_s) \wedge (i, j) \in G^{\text{SG}}(t_s) \setminus M^{\text{SG}}(t_s)\} \\ S_2(t_s) &:= \{(i, j) \in S(t_s) : (i, j) \in M^{\text{Opt}}(t_s) \wedge (i, j) \notin G^{\text{SG}}(t_s)\} \end{aligned}$$

2. For each  $(i, j) \in S_1(t_s) \cup S_2(t_s)$  do:

- a) Let  $p$  be the mapped packet closest to the tail in input queue  $VO_{ij}$  at time  $t_s$ , and let  $q \in M_L^{-1}(p)$ .
- b) Create dummy packet  $d_{ij}(t_s)$  such that  $v(d_{ij}(t_s)) = v(q)$ .
- c) Set  $M_L(q) = \emptyset$ .

**Fig. 5.** Creating dummy packets.

To fix the problem described in the previous paragraph we first rectify the definition of  $M_L$ . Looking closer into case 4 in the proof of Claim 3 it should be obvious that the condition in Lemma 1, namely  $T_1 \subseteq T_2$ , can be slightly relaxed. In fact, in order to maintain the invariant inequality defined in the proof of Claim 3, it is sufficient to remove a single mapping from one of the mapped packets in the queue whenever **Opt** transmits a packet from the queue while **SG** does not *and* the queue contains mapped packets. With that in mind, we solve the problem by creating a set of *dummy packets* (see figure 5) to handle each such instance. As a result, our  $M_L$  mapping for the input queues is now valid. We can now define an initial global weight function by  $w_\sigma(p) = \sum_{q \in M_L^{-1}(p)} v(q)$  to cover the total value lost by **SG** compared to **Opt** at the input queues, excluding the value of the dummy packets. However, we just shifted the problem further. For the analysis to work, we need to prove that the value of the dummy packets can be absorbed in the system, i.e. it can be assigned to real packets. This is done in the following step.

*Step 3: Defining the global weight function - phase 2.* The first step is to account for all dummy packets  $d_{ij}(t_s) \in S_1(t_s)$ . From now on these dummy packets will be referred to as *dummy packets of the first kind*, while dummy packets created due to  $S_2(t_s)$  will be referred to as *dummy packets of the second kind*. Recall that  $S_1(t_s) \subseteq G^{\text{SG}}(t_s)$ , and that  $S_1(t_s)$  forms a matching whose value is lower than the value of the maximum weighted matching  $M^{\text{SG}}(t_s)$ . We modify the weight function as follows. Let  $p \in M^{\text{SG}}(t_s)$  be a packet scheduled by **SG** at time step  $t_s$ . Increase  $w_\sigma(p)$  by  $v(p)$ . We thus absorbed the value of dummy packets of the first kind. Therefore, we may continue the analysis while ignoring these dummy packets.

We are now left only with the dummy packets of the second kind, i.e. packets of the form  $d_{ij}(t_s) \in S_2(t_s)$ . Denote by  $\sigma_j$  the sequence of packets scheduled by **SG** to output queue  $j$  throughout the operation of the algorithm. Further denote by  $\sigma_j^{\text{dummy}} = \cup_{t_s: (i,j) \in S_2(t_s)} d_{ij}(t_s)$  all the dummy packets of the second

kind that are destined to output queue  $O_j$ . Now recall that for every scheduling round  $t_s$ ,  $S_2(t_s) \cap G^{\text{SG}}(t_s) = \emptyset$ . By the definition of algorithm SG this means that  $v(h(VO_{ij}(t_s))) \leq \beta \cdot \min(O_j(t_s))$ . By Claim 3 it follows that  $v(d_{ij}(t_s)) \leq \beta \cdot \min(O_j(t_s))$ . Therefore, dummy packet  $d_{ij}(t_s)$  would have been rejected from queue  $O_j$  had it been sent to it. We can now apply our mapping scheme  $M_L$  to queue  $O_j$  with arrival sequence  $\sigma_j \cup \sigma_j^{\text{dummy}}$  in order to map all dummy packets in  $\sigma_j^{\text{dummy}}$  to real packets in the output queue  $O_j$ . Looking closer into the definition of  $M_L$ , this can be done while mapping each real packet in the output queue at most once. We then modify our global weight function by increasing the weight assigned to packet  $p$  by  $v(q)$ , where  $q = M_L^{-1}(p)$  is a dummy packet of the second kind. Since we have now accounted for all dummy packets of the second kind, we may consider them as absorbed in the system and ignore them.

*Step 4: Defining the global weight function - conclusion and recap.* In order to complete the definition of the weight function  $w_\sigma$  we should define the modifications in case of packet preemption at the output queues. Let packet  $p$  preempt packet  $q$  at an output queue. We increase the weight assigned to  $p$  by  $v(q) + w_\sigma(q)$ , to account for packet  $q$  and the weight assigned to it. Figure 6 summarizes the definition of the constructed global weight function  $w_\sigma$ .

**Weight function**  $w_\sigma : \text{SG}(\sigma) \rightarrow \mathbb{R}$

1. Apply mapping scheme  $M_L$  to each input queue. Add dummy packets to ensure proper operation of  $M_L$ .
2. Define  $w_\sigma(p) = \sum_{q \in M_L^{-1}(p)} v(q)$ , to cover total lost value at input queues, excluding the dummy packets.
3. Map the value of dummy packets of the first kind created at time  $t_s$  to real packets scheduled at that time.
4. Use mapping scheme  $M_L$  to map dummy packets of the second kind to real packets in output queues. Modify  $w_\sigma$  accordingly.
5. Whenever packet  $p$  preempts packet  $q$  at an output queue, modify  $w_\sigma(p) = w_\sigma(p) + w_\sigma(q) + v(q)$ .

**Fig. 6.** Construction of the global weight function  $w_\sigma$  - summary.

So far we have shown how to account for all lost packets through the use of the weight function. It remains to prove that the weight assigned to each packet is bounded.

*Claim.* By setting  $\beta = 1 + \sqrt{5}$ , the following holds:

1.  $w_\sigma(p) \leq 9.47 \cdot v(p)$ , for each packet  $p \in \text{SG}(\sigma) \setminus \text{Opt}(\sigma)$ .
2.  $w_\sigma(p) \leq 8.47 \cdot v(p)$ , for each packet  $p \in \text{SG}(\sigma) \cap \text{Opt}(\sigma)$ .

*Proof.* Consider a packet  $p \in \text{SG}(\sigma) \setminus \text{Opt}(\sigma)$ . Clearly, according to our analysis, packet  $p$  can be assigned with a weight of at most  $3v(p)$  before it reaches its destined output queue (weight of  $2v(p)$  while it resides in the input queue, and

an additional weight of  $v(p)$  to account for dummy packets of the first kind when it is scheduled to be transferred to the output queue). If  $p$  preempts a packet  $q$  once it arrives at the output queue, then its assigned weight increases by  $w(q) + v(q)$  (note that in this case  $v(p) > \beta \cdot v(q)$ ). Furthermore, once in the output queue,  $p$  can be assigned with the value of a dummy packet of the second kind, which is bounded by  $\beta \cdot v(p)$ . In order to ensure a steady state in the system, we require that the total weight ever assigned to a packet  $p$  does not exceed  $c \cdot v(p)$ . This condition corresponds to the following inequality:  $3 + \beta + \frac{c+1}{\beta} \leq c$ . Optimizing over  $\beta$ , we conclude that the optimum is obtained for  $\beta = 1 + \sqrt{5}$  and  $c \leq 9.47$ . Note that a packet  $p \in \text{SG}(\sigma) \cap \text{Opt}(\sigma)$  can be assigned with a weight of at most  $v(p)$  while it resides in the input queue (rather than  $2v(p)$  in the previous case), therefore its total assigned weight is at most  $8.47 \cdot v(p)$ .  $\square$

We conclude that:

$$\begin{aligned} V(\text{Opt}(\sigma)) &= V(\text{Opt}(\sigma) \cap \text{SG}(\sigma)) + V(\text{Opt}(\sigma) \setminus (\text{SG}(\sigma))) \\ &\leq V(\text{Opt}(\sigma) \cap \text{SG}(\sigma)) \\ &\quad + 8.47 \cdot V(\text{Opt}(\sigma) \cap \text{SG}(\sigma)) + 9.47 \cdot V(\text{SG}(\sigma) \setminus \text{Opt}(\sigma)) \\ &= 9.47 \cdot V(\text{SG}(\sigma)). \end{aligned}$$

This completes the proof of Theorem 1.  $\square$

## References

1. W. A. Aiello, Y. Mansour, S. Rajagopalan, and A. Rosén. Competitive queue policies for differentiated services. In *Proceedings of the IEEE INFOCOM 2000*, pages 431–440.
2. S. Albers and M. Schmidt. On the performance of greedy algorithms in packet buffering. In *Proc. 36th ACM Symp. on Theory of Computing*, 2004. To appear.
3. N. Andelman, Y. Mansour, and A. Zhu. Competitive queueing policies for QoS switches. In *Proc. 14th ACM-SIAM Symp. on Discrete Algorithms*, pages 761–770, 2003.
4. M. Andrews, B. Awerbuch, A. Fernández, J. Kleinberg, T. Leighton, and Z. Liu. Universal stability results for greedy contention-resolution protocols. In *Proc. 37th IEEE Symp. on Found. of Comp. Science*, pages 380–389, 1996.
5. Y. Azar and Y. Richter. Management of multi-queue switches in QoS networks. In *Proc. 35th ACM Symp. on Theory of Computing*, pages 82–89, 2003.
6. Y. Azar and Y. Richter. The zero-one principle for switching networks. In *Proc. 36th ACM Symp. on Theory of Computing*, 2004. To appear.
7. A. Birman, H. R. Gail, S. L. Hantler, Z. Rosberg, and M. Sidi. An optimal service policy for buffer systems. *Journal of the Association Computing Machinery (JACM)*, 42(3):641–657, 1995.
8. A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan, and D. Williamson. Adversarial queueing theory. In *Proc. 28th ACM Symp. on Theory of Computing*, pages 376–385, 1996.

9. E. L. Hahne, A. Kesselman, and Y. Mansour. Competitive buffer management for shared-memory switches. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 53–58, 2001.
10. A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in QoS switches. In *Proc. 33rd ACM Symp. on Theory of Computing*, pages 520–529, 2001.
11. A. Kesselman and Y. Mansour. Loss-bounded analysis for differentiated services. In *Proc. 12th ACM-SIAM Symp. on Discrete Algorithms*, pages 591–600, 2001.
12. A. Kesselman and A. Rosén. Scheduling policies for CIOQ switches. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 353–362, 2003.
13. Z. Lotker and B. Patt-Shamir. Nearly optimal fifo buffer management for diffserv. In *Proc. 21st ACM Symp. on Principles of Distrib. Computing*, pages 134–143, 2002.
14. M. May, J. C. Bolot, A. Jean-Marie, and C. Diot. Simple performance models of differentiated services for the internet. In *Proceedings of the IEEE INFOCOM 1999*, pages 1385–1394.

# Labeling Smart Dust<sup>\*</sup>

Vikas Bansal<sup>1</sup>, Friedhelm Meyer auf der Heide<sup>2</sup>, and Christian Sohler<sup>2</sup>

<sup>1</sup> Department of Computer Science and Engineering  
University of California at San Diego, La Jolla, CA 92093-0114, USA  
vibansal@cs.ucsd.edu

<sup>2</sup> Heinz Nixdorf Institute and Computer Science Department  
University of Paderborn, Germany  
{fmadh, csohler}@upb.de

**Abstract.** Given  $n$  distinct points  $p_1, p_2, \dots, p_n$  in the plane, the map labeling problem with four squares is to place  $n$  axis-parallel equi-sized squares  $Q_1, \dots, Q_n$  of maximum possible size such that  $p_i$  is a corner of  $Q_i$  and no two squares overlap. This problem is NP-hard and no algorithm with approximation ratio better than  $\frac{1}{2}$  exists unless  $P = NP$  [10].

In this paper, we consider a scenario where we want to visualize the information gathered by *smart dust*, i.e. by a large set of simple devices, each consisting of a sensor and a sender that can gather sensor data and send it to a central station. Our task is to label (the positions of) these sensors in a way described by the labeling problem above. Since these devices are not positioned accurately (for example, they might be dropped from an airplane), this gives rise to consider the map labeling problem under the assumption, that the positions of the points are not fixed precisely, but perturbed by random noise. In other words, we consider the *smoothed complexity* of the map labeling problem. We present an algorithm that, under such an assumption and Gaussian random noise with sufficiently large variance, has linear smoothed complexity.

## 1 Introduction

In a few years it will be possible to build autonomous devices with integrated sensors, communication and computing units, and power supply whose size is less than a cubic millimeter. A collection of these miniature *sensor* devices is commonly referred to as *smart dust*. One application of smart dust is the exploration of contaminated terrain, e.g. after a nuclear catastrophe, a firestorm, outbreak of a disease, etc. The sensors are dropped over the contaminated terrain from an aircraft. On the ground they gather information and use their communication units to build a network. Through this network, the gathered information is communicated to some central processing unit outside the contaminated terrain. The processing unit evaluates the incoming data. In this evaluation process, it may be very helpful to maintain a virtual map of the contaminated terrain and the sensors. In this map we want to display the value currently measured by each sensor.

---

<sup>\*</sup> Research is partially supported by DFG grant 872/8-2 and by the EU within the 6th Framework Programme under contract 001907 "Dynamically Evolving, Large Scale Information Systems" (DELIS).

Therefore, we want to assign to each sensor some *label* where the value is displayed. This label should be large, close to the sensor's position, and it should not intersect with other labels. These kind of labeling problems have been considered before in the context of map labeling [23]. We consider a basic map labeling problem, first studied by Formann and Wagner [10]. This problem is known to be  $NP$ -hard and there is no polynomial time approximation algorithm with an approximation factor of  $\frac{1}{2} + \epsilon$  for any  $\epsilon > 0$  unless  $P$  equals  $NP$ . In this paper, we consider this labeling problem in the context of smart dust, i.e. we take into consideration that we do not have a worst case distribution.

Typically, smart dust has been modeled by  $n$  points (the sensors) distributed uniformly in the unit square. In this paper we consider a refined model which captures more general distributions. Our refined model will be called the *drop point model*. The following observation forms the basis of the drop-point model. If we drop a single sensor from an airplane, we can calculate its landing point under ideal conditions (i.e., we know the exact speed of the aircraft, there is no wind, etc.). However, in real conditions, the true landing point will be somewhere near the ideal landing point. We follow standard assumptions and assume that, in general, the position of the true landing point is given by a Gaussian distribution around its ideal landing point. Of course, the idea of smart dust is to drop many sensors, not just a single one. But the airplane must not drop all sensors at once. A simple electronic device could be used to drop sensors at specific points (the drop-points). We identify with the drop points the ideal landing points of the sensors (and not the coordinates where they are dropped). Thus in our model, we have that each sensor (modeled by a point) is distributed according to a Gaussian distribution with variance  $\sigma^2$  around its drop-point; in other words we consider the *smoothed complexity* of the map labeling problem. Since drop-points may be set arbitrarily, we are interested in the worst case expected complexity under this distribution where the worst case is taken over the distributions of drop points and the expectation is taken over the Gaussian distribution around the drop points.

We show that the complexity of the labeling problem in the *drop point model* is  $O(n)$ , if  $\sigma^2 \geq n^{-1/20+\epsilon^*}$  for arbitrary small constant  $\epsilon^* > 0$ . As a byproduct of this work, we also show that we have linear running time in the average case.

## 1.1 Related Work

Map labeling is a fundamental problem in automated cartography concerned with the placement of labels near certain specified sites on a map satisfying certain constraints. Technical maps, where one is required to display some data or symbol near every point have become increasingly important in a variety of applications like display of radio/sensor networks, geographical information systems, etc. Among the various problems arising in map labeling, problems related to labeling a set of points in the plane have been extensively studied in recent years. Formann and Wagner [10] considered the basic map labeling problem where given  $n$  distinct points  $p_1, p_2, \dots, p_n$  in the plane, the objective is to place  $n$  axis-parallel equi-sized squares  $Q_1, Q_2, \dots, Q_n$  of maximum possible size such that  $p_i$  is a corner of  $Q_i$  and no two squares overlap. Using a reduction from  $3SAT$ , they proved the corresponding decision problem to be  $NP$ -complete and presented an approximation algorithm which finds a valid labeling of the points with squares of size at least half the optimal size. Moreover, they showed that there is no

hope to find an algorithm with a better approximation ratio unless  $P$  equals  $NP$ . In order to solve the problem exactly on small sized point sets, Kucera et. al. [14] presented two sub-exponential time exact algorithms for the decision version of the map labeling problem. They also observed that the optimization version can be reduced to solving  $O(\log n)$  decision problems using binary search on all inter-point distances. One of their algorithms has running time  $2^{O(\sqrt{n})}$  and is based on using a variant of the planar separator theorem. It was observed by Wagner and Wolff [22] that for most inputs, the approximation algorithm found labelings of size close to half times the optimal labeling size. They also developed several heuristics which were applicable to large point sets. Although these heuristics performed very well on most datasets and had almost linear running time, there was no theoretical justification for their performance.

Smoothed analysis was introduced by Spielman and Teng [21] as a hybrid between the worst-case and average-case analysis of algorithms to explain the polynomial running time of the simplex algorithm on practical inputs. They showed that the simplex algorithm with the shadow vertex pivot rule has polynomial time smoothed complexity. This work sparked a whole new line of research towards the smoothed analysis of algorithms [3,6]. The smoothed complexity of an algorithm is defined as the worst case expected running time of the algorithm over inputs in the neighborhood of a given instance, where the worst case is over all instances. Smoothed analysis helps to distinguish whether the space of problem instances contains only *isolated hard cases* or dense regions of hard instances. Smoothed competitive analysis of online algorithms [4] and smoothed motion complexity have been introduced [7]. Very recently, Beier and Voecking[5] have given a probabilistic analysis for a large class of NP-hard optimization problems, which enables a nice characterization of the average case and smoothed complexity of a problem in terms of the worst-case complexity.

## 1.2 Geometric Graph Model

Given an upper bound  $r$  on the optimal labeling size for an instance of the map labeling problem, we consider a geometric conflict graph  $G_P(2r) = (V, E)$  where the  $n$  points  $P$  (in the unit square) form the set of vertices  $V$  of the graph, and  $(p, q) \in E$  iff  $\|p - q\|_\infty \leq 2r$ . This conflict graph represents all possible conflicts that can occur between labels of size at most  $r$  of different points. The *random* geometric conflict graph is similar to the random geometric graph model  $G(n, r) = (V, E)$  where the  $n$  nodes are distributed uniformly in  $[0, 1]^d$  and there is an edge between two points if their Euclidean distance is at most  $r$ . Random geometric graphs model the spatial and connectivity constraints in various problems, and their properties such as connectivity, diameter, etc have been studied in the context of applications such as distributed wireless networks and sensor networks. [11,12,16]. There is also a lot of work on finding thresholds for various properties of random geometric graphs and the average case complexity of problems for  $n$  points distributed uniformly and independently in the unit square[1,2,8,18,19].

## 1.3 Our Contribution

We introduce the *drop-point model* for smart dust. The following observation forms the basis of our model. If we drop a single sensor from an airplane we can calculate its



landing point under ideal conditions (i.e., we know the exact speed of the aircraft, there is no wind, etc.). However, in real life we do not have ideal conditions. This means that the true landing point will be somewhere near the ideal landing point. We follow standard assumptions and assume that, in general, the position of the true landing point is given by a Gaussian distribution around its ideal landing point. So we can describe the position  $\tilde{p}_i$  of the  $i$ th point as  $\tilde{p}_i = p_i + e_i(\sigma)$  where  $p_i$  is the drop point of the  $i$ th point and  $e_i(\sigma)$  is a two dimensional vector drawn from the Gaussian distribution with (one dimensional) variance  $\sigma^2$  centered at the origin.<sup>1</sup> This means that the distribution  $\tilde{P}$  of our input point set can be described as a pair  $(P, \sigma)$  where  $P$  is the set of drop points  $P$  and  $\sigma^2$  is the variance of the corresponding one dimensional Gaussian distribution. For a given number  $n$  we are interested in the worst case expected complexity achieved by any distribution with  $|P| = n$ , where the worst case is taken over the choice of the set of drop points. We also call this the *smoothed complexity* of the problem under consideration.

In the remainder of this paper we present an exact algorithm for the map labeling problem with four squares introduced in [] (see next section for a definition). We show that for a wide variety of distributions for the input set of points, our algorithm has linear running time. These distributions include the uniform distribution in the unit cube and the drop point model with variance  $\sigma^2 \geq n^{-1/20+\epsilon^*}$ . We also sketch how to extend our algorithm and analysis for other variants of the map labeling problem.

## 2 Labeling Sensors

In this section we consider a basic (map) labeling problem, which was first examined in [10]. The decision version of this problem, which we call *map labeling with four squares (MLFS)* can be defined as follows: Given a set  $P$  of  $n$  distinct points  $p_1, p_2, \dots, p_n$  in the plane and a real  $\sigma$ , decide if one can place  $n$  axis-parallel squares  $Q_1, Q_2, \dots, Q_n$ , each of side length  $\sigma$ , such that each point  $p_i$  is the corner of square  $Q_i$  and no two squares overlap. In our algorithm, we make use of an exact algorithm by Kucera et. al. [14] that solves an instance of the MLFS problem in  $2^{O(\sqrt{n})}$  time.

### 2.1 The Algorithm

Since in the drop-point model the points may not lie in the unit cube, the algorithm first computes a bounding box of the input point set. Then this box is partitioned into a grid with side length  $r = \sqrt{1/n^{1+\beta}}$ , i.e. if the points are in the unit cube then the unit cube is partitioned into  $m = n^{1+\beta}$  grid cells for some constant  $\beta > 0$ . To reduce the space requirements and to have fast access to points within a given grid cell we use perfect hashing [17]. Then we first check, whether there is a grid cell that contains 5 or more points. If this is true then the following lemma gives an upper bound of  $r$  on the size of an optimal labeling.

**Lemma 1.** *Let  $P$  be a point set such that at least 5 points are within an axis-aligned square  $Q$  with side length  $r$ . Then an optimal labeling of  $P$  has size less than  $r$ .*

<sup>1</sup> The two dimensional Gaussian distribution is the product of two one dimensional Gaussian distributions. For simplicity we consider the variance of this one dimensional distribution.

*Proof.* Consider 5 points within the square  $Q$ . In any valid labeling each of these points is assigned exactly one label and these labels do not intersect. For a point  $p$  there are exactly 4 choices for placing the label, corresponding to the 4 quadrants centered at  $p$ . Therefore, in any labeling, the labels of at least 2 points are in the same quadrant. It is easy to see that if the label size is at least  $r$ , then the labels of 2 points from  $Q$  whose labels are in the same quadrant must intersect.  $\square$

In our analysis we later show that with high probability there exists a square containing 5 or more points. If our algorithm finds a square with side length  $r$  containing 5 or more points, we compute an approximation of the connected components of the conflict graph  $G_P(2r)$ . To do so, we consider grid cells that contain one or more points. We say that two grid cells  $C_1$  and  $C_2$  are  $r$ -close, if there exist points  $p \in C_1$  and  $q \in C_2$  such that  $\|(p, q)\|_\infty \leq 2r$ . Consider the transitive closure of the binary  $r$ -close relation over the set of all non-empty grid cells. Then a *connected grid cell  $r$ -component* is an equivalence class of this transitive closure. It is easy to see that for a fixed cell  $C$  the number of  $r$ -close grid cells is constant.

In order to approximate the connected components in  $G_P(2r)$  we compute the connected grid cell  $2r$ -components. We start with an arbitrary point and compute its grid cell  $C$ . Using our hash function we can determine all other points with the same grid cell. Since the number of  $2r$ -close cells for  $C$  is a constant, we can use a standard graph traversal (e.g., BFS or DFS) on the grid cells to compute the connected grid cell  $2r$ -component and all points  $P_C$  contained in its grid cells in time  $O(|P_C|)$ . Overall, we need  $O(n)$  time to compute this approximation of the connected components of  $G_P(2r)$ . Then we use the algorithm from [14] to compute an optimal labeling for the points in each connected grid component. By our upper bound on the label size these labellings are mutually non-intersecting and can therefore be combined to obtain an optimal labeling. To prove the expected linear running time we show that w.h.p, there is no connected grid component containing  $\Omega(\log n)$  points.

The following lemma summarizes our main technical result and is proved in Section 3.

**Lemma 2.** *Let  $\tilde{p}_i$  denote the random variable for the position of point  $p_i$  according to some distribution  $\mathcal{D}_i$  and let  $\tilde{P} = \{\tilde{p}_1, \dots, \tilde{p}_n\}$  denote the input point set. Consider a subdivision of the plane by a grid of side length  $\sqrt{1/n^{1+\beta}}$ . Let  $\alpha, \beta, \epsilon > 0$  be constants such that  $0 < \alpha < \beta < \frac{1}{20} + \frac{3\alpha}{5} - 5\epsilon$  is true, and let  $c_1, c_7 > 0$  be constants. If*

- $\Pr[\tilde{p}_i \text{ is in the unit cube}] \geq c_1$  and
- $\Pr[\tilde{p}_i \text{ is in cell } C] \leq \frac{1}{n^{1+\alpha}}$  for every cell  $C$  of the grid
- $\Pr[|\tilde{P}| > n^{c_7}] \leq 2^{-n}$  for some constant  $c_7$

*then algorithm FASTLABELING computes an optimal Labeling of  $\tilde{P}$  in  $O(n)$  expected time.*

Using this lemma, we obtain the following result for the smoothed complexity of the MLFS problem in the drop point model.

**Theorem 1.** *Let  $(P, \sigma)$  be a set of  $n$  points distributed according to the drop-point model and let  $\sigma^2 \geq n^{-1/20+\epsilon^*}$  for an arbitrary constant  $\epsilon^*$ . Then algorithm FASTLABELING*

**FASTLABELING( $P$ )**

1. Compute a bounding box  $B$  of  $P$
2. **if** there is  $p \in P$  with  $|p| > n^{c_7}$  **then**
3.     compute an optimal labeling using the  $2^{O(\sqrt{n})}$  exact algorithm.
4.     Subdivide  $B$  into a grid with side length  $r = \sqrt{1/n^{1+\beta}}$
5.     Pick perfect hash function  $h$  mapping the grid cells to  $\{1, \dots, 2n\}$
6.     Initialize linear lists  $L_i$  for  $1 \leq i \leq 2n$
7.     **for each** point  $p \in P$  **do**
8.         Let  $\ell$  denote the number of the square containing  $p$
9.         Store  $p$  in list  $L_{h(\ell)}$
10.    **if** some  $L_i$  contains more than 5 items in the same square **then**
11.         Determine approximate connected components of graph  $G_P(2r)$
12.         **for each** connected component  $C$  **do**
13.             Compute an optimal labeling for  $C$ .
14.    **else** compute an optimal labeling using the  $2^{O(\sqrt{n})}$  exact algorithm.

**Fig. 1.** Exact Algorithm for Map Labeling with four squares

computes an optimal labeling for  $\tilde{P}$  in  $O(n)$  expected time (where the expectation is taken over the input distribution).

*Proof.* For the drop point model the probability that a point  $\tilde{p}_i$  is contained in grid cell  $C$  is maximized when  $\tilde{p}_i$  is the center of  $C$ . For given variance  $\sigma^2$  we have in this case

$$\begin{aligned}
 \Pr[\tilde{p}_i \text{ is in cell } C] &\leq \left( \int_{-r/2}^{r/2} \sqrt{\frac{1}{2\pi\sigma^2}} \cdot e^{-\frac{x^2}{2\sigma^2}} dx \right)^2 \\
 &\leq \left( \frac{1}{\sqrt{n^{1+\beta}}} \cdot \sqrt{\frac{1}{2\pi\sigma^2}} \right)^2 \leq \left( \frac{1}{\sigma \cdot \sqrt{n^{1+\beta}}} \right)^2
 \end{aligned}$$

The second condition in Lemma 2 requires that  $\Pr[\tilde{p}_i \text{ is in cell } C] \leq \frac{1}{n^{1+\alpha}}$  for every grid cell  $C$ . This implies that  $\frac{1}{\sigma^2 \cdot n^{1+\beta}} \leq \frac{1}{n^{1+\alpha}}$ . Since we want to minimize  $\sigma$ , we have to maximize  $\beta - \alpha$ . It can be easily seen that we can achieve  $\beta - \alpha = 1/20 - \epsilon'$  for an arbitrary small constant  $\epsilon' > 0$ . This implies that  $\sigma^2 \geq n^{-1/20+\epsilon^*}$  can be achieved for an arbitrary small constant  $\epsilon^* > 0$ . As one can easily verify the other two condition of Lemma 2 are also satisfied and hence the result holds.  $\square$

The following result for the average case complexity of the MLFS problem follows immediately from Lemma 2.

**Theorem 2.** *Let  $P$  be a set of  $n$  points distributed uniformly in the unit cube. Then algorithm FASTLABELING computes an optimal labeling for  $P$  in  $O(n)$  expected time (where the expectation is taken over the input distribution).*  $\square$

### 3 Analysis

As mentioned before, we have to prove two main technical lemmas. First we show that with high probability at least one grid cell contains 5 or more points. Then we prove that with high probability the number of the points within a connected grid cell component is  $O(\log n)$ . Finally, we combine these two results to get our bound on the running time.

#### 3.1 An Upper Bound on Optimal Label Size

We assume that the point set  $\tilde{P}$  is drawn according to a probability distribution that satisfies the three assumptions in Lemma 2. That is, for each point  $\tilde{p}_i$  in  $\tilde{P}$  we have  $\Pr[\tilde{p}_i \text{ is in the unit cube}] \geq c_1$ ,  $\Pr[\tilde{p}_i \text{ is in cell } C] \leq \frac{1}{n^{1+\alpha}}$  for every cell  $C$  of a grid of side length  $r = \sqrt{1/n^{1+\beta}}$  and certain constants  $\beta > \alpha > 0$ , and  $\Pr[|\tilde{p}_i| > n^{c_7}] \leq 2^{-n}$  for some constant  $c_7$ . We denote by  $L_n^*$  the random variable for the size of an optimal solution for the MLFS problem for the point set  $\tilde{P}$ . We consider only the  $m = n^{1+\beta}$  squares that cover the unit cube. We show that, with high probability, one of these cells contains at least 5 points. Using Lemma 1 this implies that with high probability we have  $L_n^* \leq r$ . We introduce some definitions in order to prove this result.

**Definition 1.** A grid cell  $C$  is called  $\gamma$ -important for point  $\tilde{p}_i$ , if

$$\Pr[\tilde{p}_i \text{ is in Cell } C] \geq \frac{1}{n^{1+\gamma}} .$$

**Definition 2.** A grid cell is  $(\gamma, \delta)$ -heavy, if it is  $\gamma$ -important for more than  $n^{1-\delta}$  points.

Next, we show that there are many  $(\gamma, \delta)$ -heavy cells, if  $\gamma > \beta$  and  $\delta > \beta - \alpha$ .

**Lemma 3.** Let  $\gamma > \beta$ ,  $\delta > \beta - \alpha$  and  $\Pr[\tilde{p}_i \text{ is in the unit cube}] \geq c_1$ . Then the number of  $(\gamma, \delta)$ -heavy cells is at least  $c_1 \cdot n^{1+\alpha}/2$ .

*Proof.* In the following we consider only the grid cells in the unit cube. We know that the sum of the probabilities  $\Pr[\tilde{p}_i \text{ is in the unit cube}]$  over the  $n$  points, is at least  $c_1 \cdot n$ . Now assume there are less than  $c_1 \cdot n^{1+\alpha}/2$  grid cells that are  $(\gamma, \delta)$ -heavy. Then we observe that

$$\sum_{\text{heavy cell } C} \sum_{i=1}^n \Pr[\tilde{p}_i \text{ is in cell } C] \leq \frac{c_1 \cdot n^{1+\alpha}}{2} \cdot \frac{1}{n^{1+\alpha}} \cdot n = \frac{c_1 n}{2} ,$$

where the summation is over all  $(\gamma, \delta)$ -heavy cells in the unit cube. Further we use that

$$\sum_{C \text{ not heavy}} \sum_{i=1}^n \Pr[\tilde{p}_i \text{ is in cell } C] \leq n^{1+\beta} \cdot n^{1-\delta} \cdot \frac{1}{n^{1+\alpha}} + n^{1+\beta} \cdot \frac{1}{n^{1+\gamma}} = o(n) ,$$

where the sum runs over all cells in the unit cube that are not  $(\gamma, \delta)$ -heavy. It follows that the overall sum of the probabilities  $\Pr[\tilde{p}_i \text{ is in the unit cube}]$  is at most  $\frac{c_1 n}{2} + o(n)$ , which is a contradiction.  $\square$

We are now ready to prove an upper bound on the optimal label size.

**Lemma 4.**  $L_n^* \leq r$  holds with probability at least  $1 - \exp(-c_3 \cdot n^{1+\alpha-5\gamma-5\delta})$ , where  $\gamma > \beta > \alpha$  and  $\delta > \beta - \alpha$  and  $c_3$  is a constant.

*Proof.* We show that the probability that there is no cell containing at least 5 points is very small. Let  $B_l$  be the random variable that denotes the number of points in cell  $l$ ,  $1 \leq l \leq m$ , where  $m = n^{1+\beta}$ . The random variables  $(B_1, \dots, B_m)$  correspond to the process of distributing  $n$  balls into  $m$  bins according to some probability distribution and satisfy the negative regression property [15].

Let  $H$  denote the set of  $(\gamma, \delta)$ -heavy grid cells where  $|H| \geq c_1 \cdot n^{1+\alpha}/2$ . Since a cell  $j \in H$  is  $(\gamma, \delta)$ -heavy for at least  $n^{1-\delta}$  points, we have the following bound:

$$\Pr[B_j \geq k] \geq \Pr[B_j = k] \geq \binom{n^{1-\delta}}{k} \frac{1}{(n^{1+\gamma})^k} \left(1 - \frac{1}{n^{1+\gamma}}\right)^{n^{1-\delta}-k}$$

For large enough  $n$  we have  $(1 - \frac{1}{n^{1+\gamma}})^{n^{1-\delta}-k} \geq (1 - \frac{1}{n})^n \geq \frac{1}{4}$ . Hence

$$\Pr[B_j < k] \leq 1 - \binom{n^{1-\delta}}{k} \frac{1}{4 \cdot (n^{1+\gamma})^k}.$$

For  $k = 5$ , we have

$$\prod_{j \in H} \Pr[B_j < 5] \leq \left(1 - \binom{n^{1-\delta}}{5} \frac{1}{4 \cdot (n^{1+\gamma})^5}\right)^{|H|} \quad (1)$$

By negative regression it follows that for  $t_j = 5$  ( $1 \leq j \leq m$ )

$$\Pr[B_1 < 5, B_2 < 5, \dots, B_m < 5] \leq \prod_{j \in [m]} \Pr[B_j < 5] \leq \prod_{j \in H} \Pr[B_j < 5] \quad (2)$$

Combining (1) and (2), we obtain

$$\Pr[\text{there is no cell with } \geq 5 \text{ points}] \leq \exp(-c_3 \cdot n^{1+\alpha-5\gamma-5\delta})$$

for some constant  $c_3$ . □

### 3.2 Connected Grid Cell Components

For the second part of our proof we show that with high probability no connected  $2r$ -component of size  $\omega(\log n)$  exists. As we already pointed out in Section 2.1, this implies that the connected components of graph  $G_{\tilde{P}}(2r)$  have size  $O(\log n)$  w.h.p.. Finally, it suffices to solve the map labeling problem for the connected grid cell  $2r$ -components as there are no conflicts between points in different  $2r$ -components.

We consider an arbitrary point  $\tilde{p}_i$  and we want to determine the number of different sets  $S$  of  $s$  grid cells that possibly form a connected  $2r$ -component containing  $\tilde{p}_i$ , i.e. for any two cells in  $S$  there is a path using only cells in  $S$  such that two subsequent cells

on the path are  $2r$ -close. We call a set  $S$  that satisfies this condition  $2r$ -close. We now assume that all other points are unlabeled. To count the number of distinct  $2r$ -close sets of cells containing  $\tilde{p}_i$  we use a depth first search of the component and we encode this DFS by a string over an alphabet of constant size  $c_4$ . If we can show that any DFS of a  $2r$ -close set of cells of size  $s$  can be encoded by a string of length at most  $k$  then we know that the number of these sets is at most  $c_4^k$ .

**Lemma 5.** *The number of combinatorially distinct  $2r$ -close sets of size  $s$  containing  $\tilde{p}_i$  is at most  $c_4^{2s}$ .*

*Proof.* We consider a DFS starting with the cell containing  $\tilde{p}_i$ . As we already observed the number of cells that are  $2r$ -close to a given cell is a constant. Let  $c_4 - 1$  be this constant. We want to use an alphabet of size  $c_4$ . Thus for a given cell we can use symbols 1 to  $c_4 - 1$  to identify all  $2r$ -close cells. The remaining symbol is used as a special symbol. Now we can encode a DFS walk in the following way. We start at the cell containing  $\tilde{p}_i$ . Then the DFS chooses an unvisited  $2r$ -close cell that contains a point from  $\tilde{P}$ . We encode this cell by the corresponding symbol. We continue this process recursively like a standard DFS. At a certain point of time there is no unvisited  $2r$ -close cell. Then we use our special symbol to indicate that the algorithm has to backtrack.

Our string has length at most  $2s$  because we use the special symbol at most once for each cell. For each other symbol a previously unvisited cell is visited. Therefore, the length of our string is at most  $2s$  and the lemma follows.  $\square$

Our next step is to consider the probability that the cells encoded by a fixed given string form a connected  $2r$ -component containing  $\tilde{p}_i$  and  $\ell$  other points. Let us assume that the given string encodes a connected set of cells  $S$  of size  $k \leq \ell$ . We ignore the fact that each of these cells must contain at least one point. The overall number of points contained in our set of cells is exactly  $\ell$ . We have

$$\Pr[\text{there are } \ell \text{ other points in } S] \leq \binom{n}{\ell} \cdot \left( \frac{k}{n^{1+\alpha}} \right)^\ell \leq \left( \frac{n \cdot e \cdot k}{\ell \cdot n^{1+\alpha}} \right)^\ell.$$

Now we use this formula and combine it with Lemma 5 to obtain

**Lemma 6.** *The probability that point  $\tilde{p}_i$  is in a connected grid cell  $2r$ -component containing  $\ell$  other points is at most  $c_5^\ell \cdot n^{-\ell \cdot \alpha}$  for certain constant  $c_5$ .*

*Proof.* From Lemma 5 we know that the number of candidates for combinatorially distinct  $2r$ -components with  $s$  cells that contain  $\tilde{p}_i$  is at most  $c_4^{2s}$ . It follows

$$\begin{aligned} \Pr[\exists \text{ connected } 2r\text{-component with } \ell \text{ points}] &\leq \sum_{1 \leq i \leq \ell} (c_4^{2i}) \left( \frac{n \cdot e \cdot i}{\ell \cdot n^{1+\alpha}} \right)^\ell \\ &\leq c_5^\ell \cdot n^{-\ell \cdot \alpha} \end{aligned}$$

for some constant  $c_5$ .  $\square$

### 3.3 Running Time

If there exists a point  $\tilde{p}_i$  with  $|\tilde{p}_i| > n^{c_7}$  then we compute the labeling using the algorithm by Kucera et al.[14]. We use perfect hashing to store all points. The key of each point is the (number of) the grid cell it is contained in. Since  $|\tilde{p}_i| \leq n^{c_7}$  for all  $1 \leq i \leq n$ , we know that the number of cells is bounded by some polynomial in  $n$ . Then we check for each point, if it is contained in the unit cube. For all points in the unit cube we use the hash function to check, whether there are 5 or more points in the corresponding grid cell. This can be done in  $O(n)$  time. Then we compute the connected grid cell  $2r$ -components. To do this, we start at an arbitrary point and start a DFS of the  $2r$ -close grid cells. We mark all visited points. This process can also be implemented in  $O(n)$  time using our hash function. The expected running time can be calculated using the fact that with probability at most  $\exp(-c_3 \cdot n^{1+\alpha-5\gamma-5\delta})$  there is no grid cell containing 5 or more points. In this case we use the  $2^{O(\sqrt{n})}$  time exact algorithm. We also need that the time to process all components of size  $\ell$  is at most  $\frac{n}{\ell} \cdot c_6^\ell$  for some constant  $c_6$ . Let  $T(n)$  denote the expected run time of our algorithm. We get

$$T(n) \leq O(n) + 2^{O(\sqrt{n})} \cdot (2^{-n} + \exp(-c_3 \cdot n^{1+\alpha-5\gamma-5\delta})) + \sum_{1 \leq \ell \leq n} n \cdot \frac{c_6^\ell}{\ell} \cdot c_5^\ell \cdot n^{-\ell \cdot \alpha}.$$

For  $1 + \alpha - 5\gamma - 5\delta > 1/2$  and  $\alpha > 0$  we have that  $T(n) = O(n)$ . Since we also have to satisfy the inequalities  $\alpha < \beta < \gamma$  and  $\delta > \beta - \alpha$  we get running time  $O(n)$ , if the following condition can be satisfied for some  $\epsilon > 0$ :  $0 < \alpha < \beta < \frac{1}{20} + \frac{3\alpha}{5} - 5\epsilon$ . In this case we can choose  $\gamma = \beta + \epsilon$  and all conditions are satisfied. This completes the proof of Lemma 2.

## 4 Analysis for Map Labeling with Sliding Labels

Starting from the study of the basic map labeling problem with four squares by Formann and Wagner[10], exact and approximation algorithms have been obtained for various map labeling problems concerned with labeling points with polygons of different shapes and in various possible orientations. (see for e.g. [9],[13]) In this section, we show how to extend the analysis for the basic map labeling problem for the problem of labeling points with uniform axis-parallel squares in the slider model, where the point can be placed anywhere on the boundary of the square associated with it. The map labeling problem for labeling points with sliding labels is as follows:

**Definition 3 (sliding labels).** *Given a set  $P$  of  $n$  points in the plane, place  $n$  non-overlapping uniform axis-parallel squares of maximum possible size such that each point  $p_i \in P$  is associated with a unique square and lies on the boundary of its labeling square.*

This problem was shown to be NP-hard by van Kreveld et. al [13]. The best known approximation algorithm for this problem has an approximation factor of  $\frac{1}{2}$  with a running time of  $O(n^2 \log n)$  [20]. An algorithm is a *polynomial time approximation scheme* for the map labeling problem with sliding labels if it takes an additional parameter

$\epsilon$  and always computes a valid labeling of size at least  $\frac{1}{1+\epsilon}$  times the optimum labeling size, where the running time is polynomial in the representation of the set of points.

For a given value  $\delta$  and a point  $p$ , consider the square  $S_p$  of side length  $2\delta$  centered at the point  $p$ . It is easy to see that any square of side length  $\delta$  that has the point  $p$  on its boundary, lies completely within the square  $S_p$  and has two of its corners lying on one of the sides of  $S_p$ . We consider a subdivision of the square  $S_p$  by an axis-parallel grid of side length  $\frac{\delta}{\epsilon}$ . Now, consider a valid labeling for a set of points  $P$  with squares of side length  $\delta$ . For every point  $p$ , we can choose a square of size at least  $\delta(1 - \frac{\epsilon}{2})$ , such that the corners of the labeling square lie on the grid for square  $S_p$ . Observe that for a point  $p$ , there are  $\frac{8}{\epsilon}$  possible positions for placing the labeling square such that its corners lie on the grid. The next lemma gives an upper bound on the optimal labeling size for the map labeling problem with sliding labels.

**Lemma 7.** *Let  $P$  be a point set such that at least 5 points are within an axis-aligned square  $Q$  with side length  $r$ . Then an optimal labeling of  $P$  in the slider model has label size less than  $2r$ .*

The exact algorithm of Kucera et. al [14] can be extended to obtain a polynomial time approximation scheme for the map labeling problem with sliding labels.

**Lemma 8.** *There is a polynomial-time approximation scheme for the map labeling problem with sliding labels with a running time of  $2^{O(\sqrt{n} \log \frac{1}{\epsilon})}$ .*

It is easy to see that in the slider model, there can be a conflict between the labels of two points  $p$  and  $q$  only if  $d_\infty(p, q) \leq 2r$  where  $r$  is an upper bound on the optimal label size. Hence, the conflict graph  $G_P(2r)$  captures all possible conflicts given an upper bound  $r$  on the optimal label size. Therefore, the analysis in sections 3 and 4 for the MLFS problem extends without much change for the slider model. In the algorithm FASTLABELING( $P$ ), we can use the  $\epsilon$ -approximation algorithm for sliding labels instead of the  $2^{O(\sqrt{n})}$  exact algorithm.

In the running time analysis, we need to choose values for  $\alpha, \beta, \gamma$ , and  $\delta$  such that  $1 + \alpha - 5\gamma - 5\delta > 1/2 + \log_n(\log \frac{1}{\epsilon})$  and  $\alpha > 0$  in order that we have  $T(n) = O(n)$ . We also have the inequalities  $\alpha < \beta < \gamma$  and  $\delta > \beta - \alpha$ . If we can satisfy the following condition for some  $\epsilon' > 0$ :  $0 < \alpha < \beta < \frac{1}{20} + \frac{3\alpha}{5} - 5\epsilon' - \log_n(\log \frac{1}{\epsilon})$  then we can choose  $\gamma = \beta + \epsilon'$  and all conditions are satisfied.

Hence, we obtain the result that an  $\epsilon$ -approximation labeling for the map labeling problem in the slider model can be computed in  $O(n)$  expected time if the conditions of Lemma 2 hold.

## References

1. M. Appel and R. Russo. The Maximum Vertex Degree of a Graph on Uniform Points in  $[0, 1]^d$ . Adv. Appl. Prob., 567-581, 1997.
2. M. Appel and R. Russo. The connectivity of a graph on uniform points in  $[0, 1]^d$ . Statistics and Probability Letters, 60, 351-357.
3. C. Banderier, K. Mehlhorn, R. Beier. Smoothed Analysis of Three Combinatorial Problems. Proc. of the 28th International Symposium on Mathematical Foundations of Computer Science, pp. 198 - 207, 2003.



4. L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, G. Schäfer, and T. Vredeveld. Average Case and Smoothed Competitive Analysis of the Multi-Level Feedback Algorithm. Proc. of the 44th Annual IEEE Symposium on Foundations of Computer Science, pp. 462-471, 2003.
5. R. Beier, B. Vöcking. Typical Properties of Winners and Losers in Discrete Optimization. Proc. of the 36th Annual ACM Symposium on Theory of Computing, pp. 343 - 352 , 2004.
6. A. Blum, and J. Dunagan. Smoothed analysis of the perceptron algorithm for linear programming. Proc. of the 13th annual ACM-SIAM Symposium on Discrete algorithms, pp. 905 - 914, 2002.
7. V. Damerow, F. Meyer auf der Heide, H. Räcke, C. Scheideler, and C. Sohler. Smoothed Motion Complexity. Proc. of the 11th European Symposium on Algorithms, pp. 161-171, 2003.
8. J. Diaz, J. Petit, M. Serna. Random Geometric Problems on  $[0, 1]^2$ . Randomization and Approximation Techniques in Computer Science, pp. 294-306, 1998.
9. S. Doddi, M. V. Marathe, A. Mirzaian, B. M.E. Moret, and B. Zhu. Map labeling and its generalizations. Proc. of the 8th ACM-SIAM Symposium on Discrete Algorithms, pp. 148-157, 1997.
10. M. Formann and F. Wagner A packing problem with applications to lettering of maps. Proc. of the 7th Annual ACM Symposium on Computational Geometry, pp. 281-288, 1991.
11. A. Goel, S. Rai, B. Krishnamachari. Sharp thresholds for monotone properties in random geometric graphs. Proc. of the 36th Annual ACM Symposium on Theory of Computing, pp. 580-586, 2004.
12. P. Gupta and P. Kumar. Critical Power for Asymptotic Connectivity in Wireless Networks. In Stochastic Analysis, Control, Optimization and Applications: A Volume in Honor of W. H. Fleming. W. M. McEneaney, G. Yin, and Q. Zhang eds., 1998.
13. M. van Kreveld, T. Strijk, and A. Wolff. Point labeling with sliding labels. TR Comput. Geom. Theory Appl., 13:21-47, 1999.
14. L. Kucera, K. Mehlhorn, B. Preis, and E. Schwarzenegger. Exact algorithms for a geometric packing problem. In Proc. 10th Symposium on Theoretical Aspects in Computer Science, pages 317-322. , 1993.
15. D. Dubhashi and D. Ranjan. Balls and bins: a study in negative dependence. Random Structures and Algorithms, 13(2): 99-124, 1998.
16. S. Meguerdichian, F. Koushanfar, M. Potkonjak, M. B. Srivastava. Coverage Problems in wireless ad-hoc sensor networks. In Proc. of INFOCOM 2001, 1380-87, 2001.
17. K. Mehlhorn. Data Structures and Algorithms 1: Sorting and Searching. Springer, 1984.
18. S. Muthukrishnan, Gopal Pandurangan. The Bin-Covering Technique for Thresholding Random Geometric Graph Properties. DIMACS Technical Report 2003.
19. M. Penrose. On k-Connectivity for a Geometric Random Graph. Random Structures and Algorithms 15, pp. 145-164, 1999.
20. Z. Qin, B. Zhu. A Factor-2 Approximation for Labeling Points with Maximum Sliding Labels. Proc. of the 8th Scandinavian Workshop on Algorithm Theory, pp. 100-109, 2002.
21. D. Spielman and S. Teng. Smoothed Analysis of Algorithms: Why The Simplex Algorithm Usually Takes Polynomial Time. Proc. of the 33rd Annual ACM Symposium on Theory of Computing, pp. 296-305., 2001.
22. F. Wagner and A. Wolff. Map labeling heuristics: Provably good and practically useful. Technical Report B 95-04, Institut für Informatik, Freie Universität Berlin, April 1995.
23. A. Wolff. The Map-Labeling Bibliography.  
<http://i11www.ira.uka.de/map-labeling/bibliography/>.

# Graph Decomposition Lemmas and Their Role in Metric Embedding Methods

Yair Bartal\*

Department of Computer Science, Hebrew University, Jerusalem, Israel  
yair@cs.huji.ac.il

**Abstract.** We present basic graph decomposition lemmas and show how they apply as key ingredients in the probabilistic embedding theorem stating that every  $n$  point metric space probabilistically embeds in ultrametrics with distortion  $O(\log n)$  and in the proof of a similar bound for the spreading metrics paradigm in undirected graphs. This provides a unified framework for these metric methods which have numerous algorithmic applications.

## 1 Introduction

This paper presents new graph decomposition lemmas which lie at the heart of two metric-based techniques that have been the basis for many results in approximation and online algorithms in recent years: probabilistic metric embedding in ultrametrics [2] and the spreading metrics paradigm [10]. We thus provide a unified framework for these techniques.

Our main graph decomposition lemma is similar in spirit to the well-known graph decompositions of Garg, Vazirani and Yannakakis [15] and Seymour [19].

Such a decomposition finds a low diameter cluster in the graph such that the weight of the cut created is small respect to the weight of the cluster. These type of decompositions have been extremely useful in the theory of approximation algorithms, e.g. for problems such as minimum multi-cut, balanced separators, and feedback arc-set. Our new decomposition is essentially based on the decomposition of [15] performed in a careful manner so as to achieve a more refined bound on the ratio between the cut and the cluster's weight. The graph decomposition lemma is presented in Section 2.

We also give a new probabilistic decomposition lemma. Probabilistic partitions, introduced in [2], are partitions of the graph into low diameter clusters with small probability of cutting any edge. Such partitions have been used in the context of probabilistic embeddings and are shown over time to have a yet more significant role in the theory of metric embeddings. Here, we present a variation on this idea. Instead of partitioning the entire graph we focus on constructing one low diameter cluster, as in the previous lemma case. Roughly, the decomposition has the property that the probability an edge is cut is small with respect

---

\* Supported in part by a grant from the Israeli National Science Foundation (195/02).

to the probability that it is included in the cluster. The lemma is essentially based on the decomposition of [2] performed in a careful manner so as to achieve a more refined bound on the ratio between these probabilities. The probabilistic decomposition lemma is presented in Section 4.

Probabilistically embedding metric spaces in ultrametrics (and hierarchically well-separated trees) was introduced in [2]. The original metric space is embedded in a randomly chosen (dominating) ultrametric so that the distortion is kept low in expectation. Such an embedding is extremely useful for algorithmic application since it reduces the problem of obtaining an approximation algorithm for the original space to obtaining an algorithm for ultrametrics, a task which is often considerably easier. It has implications for embeddings in low dimensional normed spaces as well [7]. In [2] we have introduced probabilistic partitions and proved that such a partition gives a probabilistic embedding in ultrametrics. In [3] we have proved that every metric space on  $n$  points can be probabilistically embedded in ultrametrics with distortion  $O(\log n \log \log n)$ , based on the technique of [19]. Recently, the bound was improved to  $O(\log n)$  [13] by providing an improved probabilistic partition, which is based on techniques from [9,12]. This bound is tight [2]. The theorem has been used in many applications, including metric labelling [17], bulk-at-buy network design [1], group Steiner tree [14], min-sum clustering [5], metrical task systems [4,11], etc. (see the survey [16] for a more comprehensive summary of applications.) The approximation algorithm for some of these problems can be made deterministic via methods of [8] by using the dual problem to probabilistic embedding, which we call distributional embedding: given a distribution over the pairs in the metric space construct a single ultrametric as to approximate the expected distance. It follows from duality that the best solutions for these two problems have the same distortion bound. Moreover, it is shown in [3,8] that a solution to the dual problem can be used to construct an efficient probabilistic embedding. In [13] such a dual solution is obtained by applying derandomization techniques to their probabilistic algorithm. Here, we use our graph decomposition lemma to give a simple direct construction of a distributional embedding in ultrametrics with  $O(\log n)$  distortion. This implies simpler approximation algorithms in the applications. This is done in Section 3.

We next turn to the probabilistic embedding version, and show how our probabilistic decomposition lemma, together with techniques from [3], yield a simple direct efficient construction of probabilistic embedding in ultrametrics with  $O(\log n)$  distortion. This appears in Section 4.

A related notion of embedding is that of multi-embeddings introduced in [6]. We note that our graph decomposition lemma applies to obtain improved results for multi-embeddings in ultrametrics as well.

Spreading metrics were introduced in [10]. They consider minimization problems defined in a graph  $G$  with capacities and assume the existence of a spreading metric which provides a lower bound on the cost of a solution for the problem as well as some guaranty which relates the distances to the cost on smaller parts of the problem. In the context of this paper we restrict the graph  $G$  to be undi-

rected. This method has been applied to several problems, including linear arrangement, balanced partitions, etc. An approximation ratio of  $O(\log n \log \log n)$  was given for these problems [10], based on the technique of [19]. Most of these were later improved in [18] to  $O(\log n)$ . However, their method was not general and excluded, in particular, the problem of embedding graphs in  $d$ -dimensional meshes, where we would like to minimize the average distance in the embedding. In this paper we improve the generic spreading metric bound to  $O(\log n)$ . We note that the spreading metrics paradigm applies to some other problems in directed graphs, including directed feedback arc-set. The directed case remains a challenging open problem. We discuss the application of our graph decomposition lemma to the spreading metrics paradigm in Section 5.

## 2 Graph Decompositions

### 2.1 Decomposition Lemma

Let  $G = (V, E)$  be an undirected graph with two weight functions  $c, x : E \rightarrow \mathbb{R}^+$ . We interpret  $x(e)$  to be the length of the edge  $e$ , and the distance between pairs of vertices  $u, v$  in the graph, denote  $d(u, v)$ , is determined by the length of the shortest path between them. Given a subgraph  $H = (V_H, E_H)$  of  $G$ , let  $d_H$  denote the distance in  $H$ , let  $\Delta(H)$  denote the diameter of  $H$ , and  $\Delta = \Delta(G)$ . We define the volume of  $H$ ,  $\phi(H) = \sum_{e \in E_H} x(e)c(e)$ .

Given a subset  $S \subseteq V$ ,  $G(S)$  denotes the subgraph of  $G$  induced by  $S$ . Given partition  $(S, \bar{S})$  let  $\Gamma(S) = \{(u, v) \in E; u \in S, v \in \bar{S}\}$  and  $\text{cut}(S) = \sum_{e \in \Gamma(S)} c(e)$ .

For a vertex  $v$  and  $r \geq 0$ , the ball at radius  $r$  around  $v$  is defined as  $B(v, r) = \{u \in V | d(u, v) \leq r\}$ . Let  $S = B(v, r)$ . Define

$$\bar{\phi}(S) = \bar{\phi}(v, r) = \sum_{e=(u,w); u,w \in S} x(e)c(e) + \sum_{e=(u,w) \in \Gamma(S)} c(e)(r - d(v, u)).$$

Given a subgraph  $H$ , we can similarly define  $\bar{\phi}_H$  with respect to the subgraph  $H$ . Define the spherical-volume of  $H$ ,

$$\phi^*(H) = \max_{v \in H} \bar{\phi}_H(v, \Delta(H)/4).$$

**Lemma 1.** *Given a graph  $G$ , there exist a partition  $(S, \bar{S})$  of  $G$ , where  $S$  is a ball and*

$$\text{cut}(S) \leq \frac{8 \ln(\phi^*(G)/\phi^*(G(S)))}{\Delta(G)} \cdot \bar{\phi}(S).$$

*Proof.* Let  $v$  be a node that minimizes the ratio  $\bar{\phi}(v, \Delta/4)/\bar{\phi}(v, \Delta/8)$ . We will choose  $S = B(v, r)$  for some  $r \in [\Delta/8, \Delta/4]$ . We apply a standard argument,

similar to that of [15]. First observe that  $\frac{d\bar{\phi}(v,r)}{dr} = \text{cut}(B(v,r))$ . Therefore there exist  $r$  such that

$$\begin{aligned} \text{cut}(B(v,r))/\bar{\phi}(v,r) &\leq \left( \int_{r=\Delta/8}^{\Delta/4} \frac{d\bar{\phi}(v,r)}{\bar{\phi}(v,r)} \right) / \left( \frac{\Delta(G)}{8} \right) \\ &= \frac{\ln(\bar{\phi}(v, \Delta/4)/\bar{\phi}(v, \Delta/8))}{\Delta(G)/8}. \end{aligned}$$

Now, let  $u$  be the node that maximizes  $\bar{\phi}_S(u, \Delta(G(S))/4)$ . Since  $\Delta(G(S)) \leq \Delta/2$  we have that  $\phi^*(G(S)) \leq \bar{\phi}(u, \Delta/8)$ . Recall that  $\phi^*(G) \geq \bar{\phi}(u, \Delta/4)$ . By the choice of  $v$  we have

$$\begin{aligned} \text{cut}(S) &\leq \frac{8 \ln(\bar{\phi}(u, \Delta/4)/\bar{\phi}(u, \Delta/8))}{\Delta(G)} \cdot \bar{\phi}(v, r) \\ &\leq \frac{8 \ln(\phi^*(G)/\phi^*(G(S)))}{\Delta(G)} \cdot \bar{\phi}(S). \end{aligned}$$

□

## 2.2 Recursive Application

The decomposition comes useful where it is applied recursively. We show in Sections 3 and 5 that our applications posses a cost function  $\alpha$  over subgraphs  $\hat{G}$  of  $G$  which is nonnegative, 0 on singletons, and obeys the recursion:

$$\alpha(\hat{G}) \leq \alpha(\hat{G}(S)) + \alpha(\hat{G}(\bar{S})) + \Delta(\hat{G}) \cdot \text{cut}(S). \quad (1)$$

**Lemma 2.** *The function defined by (1) obeys  $\alpha(G) \leq O(\log(\phi/\phi_0)) \cdot \phi(G)$  where  $\phi = \phi(G)$  and  $\phi_0$  is the minimum value of  $\phi(\hat{G})$  on non-singleton subgraphs  $\hat{G}$ .*

*Proof.* We prove by induction on the size of subgraphs  $\hat{G}$  of  $G$  that  $\alpha(\hat{G}) \leq 8 \ln(\phi^*(\hat{G})/\phi_0) \cdot \phi(\hat{G})$ . Let  $\hat{G}$  be a subgraph of  $G$ . Using lemma 1 for  $\hat{G}$  we obtain  $S = B(v, r)$  so that

$$\begin{aligned} \alpha(\hat{G}) &\leq \alpha(\hat{G}(S)) + \alpha(\hat{G}(\bar{S})) + \Delta(\hat{G}) \cdot \text{cut}(S) \\ &\leq 8 \ln(\phi^*(\hat{G}(S))/\phi_0) \cdot \phi(\hat{G}(S)) + 8 \ln(\phi^*(\hat{G}(\bar{S}))/\phi_0) \cdot \phi(\hat{G}(\bar{S})) \\ &\quad + 8 \ln(\phi^*(\hat{G})/\phi^*(\hat{G}(S))) \cdot \bar{\phi}(v, r) \\ &\leq 8 \ln(\phi^*(\hat{G})/\phi_0) \cdot \bar{\phi}(v, r) + 8 \ln(\phi^*(\hat{G})/\phi_0) \cdot \phi(\hat{G}(\bar{S})) \\ &\leq 8 \ln(\phi^*(\hat{G})/\phi_0) \cdot \phi(\hat{G}). \end{aligned}$$

The result follows since  $\phi^*(\hat{G}) \leq \phi(\hat{G})$ . □

To obtain a bound depending only on  $n$ , we again apply a standard argument from [15]: modify the process slightly by associating a volume  $\phi(G)/n$  with nodes. This ensures that  $\phi_0 \geq \phi(G)/n$ .

**Lemma 3.** *The function defined by (1) using the modified procedure obeys  $\alpha(G) \leq O(\log n) \cdot \phi(G)$ .*

### 3 Embedding in Ultrametrics

**Definition 1.** An ultrametric ( $HST^1$ ) is a metric space whose elements are the leaves of a rooted tree  $T$ . Each vertex  $u \in T$  is associated with a label  $\Delta(u) \geq 0$  such that  $\Delta(u) = 0$  iff  $u$  is a leaf of  $T$ . It is required that if  $u$  is a child of  $v$  then  $\Delta(u) \leq \Delta(v)$ . The distance between two leaves  $x, y \in T$  is defined as  $\Delta(\text{lca}(x, y))$ , where  $\text{lca}(x, y)$  is the least common ancestor of  $x$  and  $y$  in  $T$ .

We study the set of metric spaces defined over an  $n$ -point set  $V$ . Given such a metric space  $M$ , the distance between  $u$  and  $v$  in  $V$  is denoted  $d_M(u, v)$ . It is useful to identify the metric space with a complete weighted graph  $G = (V, E)$ , where edge weights  $x(e)$  are distances in  $M$ .

The following definitions are from [2].

**Definition 2.** A metric space  $N$  dominates another metric space  $M$  if for every  $u, v \in V$ ,  $d_N(u, v) \geq d_M(u, v)$ .

**Definition 3.** A metric space  $M$   $\alpha$ -probabilistically embeds in a class of metric spaces  $\mathcal{N}$  if every  $N \in \mathcal{N}$  dominates  $M$  and there exists a distribution  $\mathcal{D}$  over  $\mathcal{N}$  such that for every  $u, v \in V$ :

$$\mathbb{E}[d_N(u, v)] \leq \alpha d_M(u, v).$$

The following notion serves as a dual to probabilistic embedding [3,8].

**Definition 4.** A metric space  $M$   $\alpha$ -distributionally embeds in a class of metric spaces  $\mathcal{N}$  if every  $N \in \mathcal{N}$  dominates  $M$  and given any weight function  $c : V^2 \rightarrow \mathbb{R}^+$ , there exists  $N \in \mathcal{N}$  satisfying:

$$\sum_{u, v \in V^2} c(u, v) d_N(u, v) \leq \alpha \sum_{u, v \in V^2} c(u, v) d_M(u, v).$$

It follows from duality considerations (see [3,8]) that a metric space  $\alpha$ -distributionally embeds in  $\mathcal{N}$  iff it  $\alpha$ -probabilistically embeds in  $\mathcal{N}$ .

**Theorem 1.** Any metric space on  $n$  points  $O(\log n)$ -distributionally embeds in an ultrametric.

*Proof.* We build an ultrametric recursively as follows: use the decomposition described in Section 2 to obtain a partition of the graph  $(S, \bar{S})$ . Run the algorithm on each part recursively, obtaining ultrametrics ( $HST$ s)  $T_S$  and  $T_{\bar{S}}$  respectively. An ultrametric ( $HST$ )  $T$  is constructed by creating a root  $r$  labelled  $\Delta$  with two children at which we root the trees  $T_S$  and  $T_{\bar{S}}$ . Define  $\text{cost}(T) = \sum_{(u, v) \in E} c(u, v) d_T(u, v)$ . Then

$$\text{cost}(T) = \text{cost}(T_S) + \text{cost}(T_{\bar{S}}) + \Delta \cdot \text{cut}(S).$$

The theorem now follows from Lemma 3. □

<sup>1</sup> A  $k$ -HST [2] is defined similarly while requiring that  $\Delta(u) \leq \Delta(v)/k$ . Any ultrametric  $k$ -embeds [3] and  $O(k/\log k)$ -probabilistically embeds [5] in a  $k$ -HST.

## 4 Probabilistic Embedding

Theorem 1 implies the basic result on probabilistic embedding of arbitrary metrics in ultrametrics. Moreover, it is shown in [3,8] how to efficiently construct the probabilistic embedding. In this section we combine ideas from our new graph decomposition together with ideas from [2,3] to obtain the following probabilistic decomposition lemma, which is then applied to give a simple and efficient direct construction of a probabilistic embedding in ultrametrics.

### 4.1 Probabilistic Decomposition Lemma

Let  $R = \Delta/C$ , where  $C = 128$ . Define  $\lambda(x) = |B(x, 16R)|/|B(x, R)|$ . Let  $v$  be the vertex that minimizes  $\lambda(v)$ . Choose a radius  $r$  in the interval  $[2R, 4R]$  according to the distribution<sup>2</sup>  $p(r) = (\frac{\lambda(v)^2}{1-\lambda(v)^{-2}}) \frac{\ln \lambda(v)}{R} \lambda(v)^{-\frac{r}{R}}$ . Now define a partition  $(S, \bar{S})$  where  $S = B(v, r)$ .

The distribution above is chosen to satisfy the following relation:

**Lemma 4.**

$$\Pr[(x, y) \in \Gamma(S)] \leq [(1 - \Pr[x, y \in \bar{S}]) \cdot \ln \lambda(x) + \lambda(v)^{-1}] \cdot \frac{d(x, y)}{R}.$$

*Proof.* Assume w.l.o.g. that  $y$  is closer to  $v$  than  $x$ . We have:

$$\begin{aligned} \Pr[(x, y) \in \Gamma(S)] &= \int_{d(v, y)}^{d(v, x)} p(r) dr \\ &= \left(\frac{\lambda(v)^2}{1-\lambda(v)^{-2}}\right) \lambda(v)^{-\frac{d(v, y)}{R}} (1 - \lambda(v)^{-\frac{d(v, x)-d(v, y)}{R}}) \\ &\leq \left(\frac{\lambda(v)^2}{1-\lambda(v)^{-2}}\right) \lambda(v)^{-\frac{d(v, y)}{R}} \cdot \frac{d(x, y)}{R} \ln \lambda(v), \end{aligned} \quad (2)$$

$$\begin{aligned} 1 - \Pr[x, y \in \bar{S}] &= \int_{d(v, y)}^{4R} p(r) dr \\ &= \left(\frac{\lambda(v)^2}{1-\lambda(v)^{-2}}\right) (\lambda(v)^{-\frac{d(v, y)}{R}} - \lambda(v)^{-4}). \end{aligned} \quad (3)$$

Therefore we have:

$$\begin{aligned} &\Pr[(x, y) \in \Gamma(S)] - (1 - \Pr[x, y \in \bar{S}]) \cdot \frac{d(x, y)}{R} \ln \lambda(x) \\ &\leq \left(\frac{\lambda(v)^2}{1-\lambda(v)^{-2}}\right) \lambda(v)^{-4} \cdot \ln \lambda(v) \cdot \frac{d(x, y)}{R} \leq \lambda(v)^{-1} \cdot \frac{d(x, y)}{R}, \end{aligned}$$

where we have used the fact that  $\lambda(x) \geq \lambda(v)$ , and the last inequality holds for any value  $\lambda(v) \geq 1$ . This completes the proof of the lemma.  $\square$

<sup>2</sup> This is indeed a probability distribution. Note that we may assume  $\lambda(v) > 1$  otherwise the partition is trivial.

## 4.2 The Probabilistic Embedding Theorem

**Theorem 2.** *Any metric space on  $n$  points  $O(\log n)$ -probabilistically embeds in an ultrametric.*

*Proof.* We build an ultrametric recursively as follows: use the probabilistic decomposition described above to obtain a partition of the graph  $(S, \bar{S})$ . Run the algorithm on each part recursively, obtaining ultrametries (HSTs)  $T_S$  and  $T_{\bar{S}}$  respectively. An ultrametric (HST)  $T$  is constructed by creating a root  $r$  labelled  $\Delta$  with two children at which we root the trees  $T_S$  and  $T_{\bar{S}}$ .

The analysis shows that for every  $(x, y) \in E$ :

$$\frac{E[d_T(x, y)]}{C \cdot d(x, y)} \leq \ln |B(x, 16R)| + \ln |B(x, 8R)|.$$

The proof is by induction on the construction of the HST. We have

$$\begin{aligned} E[d_T(x, y)] &= \Pr[(x, y) \in \Gamma(S)]\Delta \\ &\quad + \Pr[x, y \in S] \cdot E[d_{T_S}(x, y)] + \Pr[x, y \in \bar{S}] \cdot E[d_{T_{\bar{S}}}(x, y)]. \end{aligned} \quad (4)$$

We now apply the induction hypothesis. We assume that  $\Pr[(x, y) \in \Gamma(S)] \neq 0$ , otherwise the result follows immediately from the induction hypothesis. This implies that  $B(v, 4R)$  includes  $x$  or  $y$ . We may also assume  $d(x, y) \leq R/2$ , otherwise the result is immediate. It follows that  $d(v, x) < 5R$  and so that  $B(v, R) \subseteq B(x, 8R)$ .

$$\begin{aligned} \frac{E[d_{T_S}(x, y)]}{C \cdot d(x, y)} &\leq \ln |B(x, 16R) - S| + \ln |B(x, 8R) - S| \\ &\leq \ln |B(x, 16R)| + \ln(|B(x, 8R)| - |B(v, R)|), \end{aligned} \quad (5)$$

Since  $\Delta(S) \leq 8R$  we have:

$$\frac{E[d_{T_S}(x, y)]}{C \cdot d(x, y)} \leq \ln |B(x, R)| + \ln |B(x, R/2)|,$$

and since  $B(v, 2R)$  does not include both  $x$  and  $y$ , we may assume that  $d(v, x) \geq 2R - d(x, y) \geq 3/2 \cdot R$ . Thus, we have that  $B(x, R/2) \subseteq B(x, 8R) - B(v, R)$ , which gives:

$$\frac{E[d_{T_S}(x, y)]}{C \cdot d(x, y)} \leq \ln |B(x, R)| + \ln(|B(x, 8R)| - |B(v, R)|). \quad (6)$$

We apply Lemma 4. Recall that  $\lambda(v)^{-1} = |B(v, R)|/|B(v, 16R)|$ . Since  $d(v, x) < 5R$  we have that  $B(x, 8R) \subseteq B(v, 16R)$ , hence  $\lambda(v)^{-1} \leq |B(v, R)|/|B(x, 8R)|$ . Thus the lemma implies:

$$\begin{aligned} \Pr[(x, y) \in \Gamma(S)] &\leq [(1 - \Pr[x, y \in \bar{S}]) \cdot \ln(|B(x, 16R)|/|B(x, R)|) \\ &\quad + |B(v, R)|/|B(x, 8R)|] \cdot \frac{d(x, y)}{R}. \end{aligned} \quad (7)$$



We get the inductive claim by plugging the above inequalities in (4):

$$\begin{aligned}
\frac{\mathbb{E}[d_T(x, y)]}{Cd(x, y)} &\leq (1 - \Pr[x, y \in \bar{S}]) \left[ \ln \frac{|B(x, 16R)|}{|B(x, R)|} + \ln |B(x, R)| \right] \\
&\quad + \Pr[x, y \in \bar{S}] \cdot \ln |B(x, 16R)| \\
&\quad + \left[ \ln(|B(x, 8R)| - |B(v, R)|) + \frac{|B(v, R)|}{|B(x, 8R)|} \right] \\
&\leq \ln |B(x, 16R)| + \ln |B(x, 8R)|.
\end{aligned}$$

□

## 5 Spreading Metrics

The spreading metric paradigm [10] applies to minimization problems on undirected graphs  $G = (V, E)$  with edge capacities  $c(e) \geq 1$  for  $e \in E$ . Associated is an auxiliary graph  $H$  and a scaler function on subgraphs of  $H$ . A decomposition tree  $T$  is a tree with nodes corresponding to non-overlapping subsets of  $V$ , forming a recursive partition of  $V$ . For a node  $t$  of  $T$  denote by  $V_t$  the subset at  $t$ . Associated are the subgraphs  $G_t, H_t$  induced by  $V_t$ . Let  $F_t$  be the set of edges that connect vertices that belong to different children of  $t$ , and  $c(F_t) = \sum_{e \in F_t} c(e)$ . The cost of  $T$  is  $\text{cost}(T) = \sum_{t \in T} \text{scaler}(H_t) \cdot c(F_t)$ .

**Definition 5.** A spreading metric is a function  $x : E \rightarrow \mathbb{R}^+$  satisfying:  $\sum x(e)c(e)$  is a lower bound on the optimal cost, and for any  $U \subseteq V$ , and  $H_U$  the subgraph of  $H$  induced by  $U$ ,  $\Delta(U) \geq \text{scaler}(H_U)$ .

**Theorem 3.** There exist an  $O(\log n)$  approximation for problems in the spreading metrics paradigm.

*Proof.* The algorithm is defined recursively: use the decomposition described in Section 2 to obtain a partition of the graph  $(S, \bar{S})$ . Run the algorithm on each part recursively. A tree  $T$  is constructed by creating a root  $r$  with two children at which we root the trees  $T_S$  and  $T_{\bar{S}}$ . The cost of the tree is given by

$$\begin{aligned}
\text{cost}(T) &= \text{cost}(T_S) + \text{cost}(T_{\bar{S}}) + \text{scaler}(H) \cdot \text{cut}(S) \\
&\leq \text{cost}(T_S) + \text{cost}(T_{\bar{S}}) + \Delta \cdot \text{cut}(S).
\end{aligned}$$

The result now follows from Lemma 3. □

## References

1. B. Awerbuch and Y. Azar. Buy-at-Bulk Network Design. In *Proc. 38th IEEE Symp. on Foundations of Computer Science*, 542-547, 1997.
2. Y. Bartal, Probabilistic Approximation of Metric Spaces and its Algorithmic Applications. In *Proc. of the 37rd Ann. IEEE Symp. on Foundations of Computer Science*, 184-193, 1996.

3. Y. Bartal, On Approximating Arbitrary Metrics by Tree Metrics. In *Proc. of the 30th Ann. ACM Symp. on Theory of Computing*, 161-168, 1998.
4. Y. Bartal, A. Blum, C. Burch, and A. Tomkins. A  $\text{polylog}(n)$ -Competitive Algorithm for Metrical Task Systems. In *Proc. of the 29th Ann. ACM Symp. on Theory of Computing*, 711-719, 1997.
5. Y. Bartal, M. Charikar, and D. Raz. Approximating Min-Sum Clustering in Metric Spaces. In *Proc. of the 33rd Ann. ACM Symp. on Theory of Computing*, 11-20, 2001.
6. Y. Bartal and M. Mendel. Multi-Embedding and Path Approximation of Metric Spaces. In *Proc. of the 14th ACM-SIAM Symp. on Discrete Algorithms*, 424-433, 2003.
7. Y. Bartal and M. Mendel. Dimension Reduction for ultrametrics. In *Proc. of the 15th ACM-SIAM Symp. on Discrete Algorithms*, 664-665, January 2004.
8. M. Charikar, C. Chekuri, A. Goel, S. Guha, and S. Plotkin Approximating a Finite Metric by a Small Number of Tree Metrics, IEEE Symposium on Foundations of Computer Science, 379-388, 1998.
9. G. Calinescu, H. Karloff, Y. Rabani. Approximation Algorithms for the 0-Extension Problem. In *Proc. of the 12th ACM-SIAM Symp. on Discrete Algorithms*, 8-16, 2001.
10. G. Even, J. Naor, S. Rao, and B. Scheiber. Divide-and-Conquer Approximation Algorithms via Spreading Metrics. In *Proc. of the 36th Ann. IEEE Symp. on Foundations of Computer Science*, 62-71, 1995.
11. Amos Fiat and Manor Mendel. Better Algorithms for Unfair Metrical Task Systems and Applications, In *Proc. of the 32nd Annual ACM Symposium on Theory of Computing*, 725-734, 2000.
12. J. Fakcharoenphol, C. Harrelson, S. Rao and K. Talwar. An Improved Approximation Algorithm for the 0-Extension Problem. In *Proc. of the 14th ACM-SIAM Symp. on Discrete Algorithms*, 257-265, 2003.
13. J. Fakcharoenphol, S. Rao and K. Talwar. A Tight bound on Approximating Arbitrary metrics by Tree metrics. In *Proc. of the 35th Ann. ACM Symp. on Theory of Computing*, 448-455, 2003.
14. N. Garg, G. Konjevod, and R. Ravi. A Polylogarithmic Algorithm for the Group Steiner Tree Problem. In *Journal of Algorithms*, 37(1):66-84, 2000.
15. N. Garg, V. Vazirani, and Y. Yannakakis. Approximate Max-Flow Min-(Multi) Cut Theorems and Their Applications. In *SIAM J. Computing* 25, 235-251, 1996.
16. P. Indyk. Algorithmic Applications of Low-Distortion Geometric Embeddings. In *Proc. of 42nd Symposium on Foundations of Computer Science*, 10-33, 2001.
17. J. M. Kleinberg and E. Tardos. Approximation Algorithms for Classification Problems with Pairwise Relationships: Metric Labeling and Markov Random Fields. In *Proc. of the 40th Ann. IEEE Symp. on Foundations of Computer Science*, 14-23, 1999.
18. S. Rao and A. Richa. New Approximation Algorithms for some Ordering Problems. In *Proc. 9th ACM-SIAM Symp. on Discrete Algorithms*, 211-219, 1998.
19. P.D. Seymour. Packing Directed Circuits Fractionally. In *Combinatorica*, 15(2):281-288, 1995.

# Modeling Locality: A Probabilistic Analysis of LRU and FWF\*

Luca Becchetti

University of Rome “La Sapienza”,  
becchett@dis.uniroma1.it

**Abstract.** In this paper we explore the effects of locality on the performance of paging algorithms. Traditional competitive analysis fails to explain important properties of paging assessed by practical experience. In particular, the competitive ratios of paging algorithms that are known to be efficient in practice (e.g. LRU) are as poor as those of naive heuristics (e.g. FWF). It has been recognized that the main reason for these discrepancies lies in an unsatisfactory modelling of locality of reference exhibited by real request sequences.

Following [13], we explicitly address this issue, proposing an adversarial model in which the probability of requesting a page is also a function of the page’s age. In this way, our approach allows to capture the effects of locality of reference. We consider several families of distributions and we prove that the competitive ratio of LRU becomes constant as locality increases, as expected. This result is strengthened when the distribution satisfies a further concavity/convexity property: in this case, the competitive ratio of LRU is always constant.

We also propose a family of distributions parametrized by locality of reference and we prove that the performance of FWF rapidly degrades as locality increases, while the converse happens for LRU.

We think, our results provide one contribution to explaining the behaviours of these algorithms in practice.

## 1 Introduction

Paging is the problem of managing a two-level memory consisting of a first level fast memory or *cache* and a second level slower memory, both divided into pages of equal, fixed size. We assume that the slow memory contains a universe  $\mathcal{P} = \{p_1, \dots, p_M\}$  of pages that can be requested, while the cache can host a fixed number  $k$  of pages. Normally,  $M \gg k$ .

The input to a paging algorithm is a sequence of on-line requests  $\sigma = \{\sigma(1), \dots, \sigma(n)\}$ , each specifying a page to access. The  $i$ -th request is said to have *index*  $i$ . If the page is in main memory the access operation costs nothing. Otherwise, a page fault occurs: the requested page has to be brought from main memory into cache in order to be accessed. If the cache is full, one of the pages

---

\* Partially supported by the EU projects AMORE, ALCOM-FT, COSIN and by the MIUR Project “Resource Allocation in Real Networks”.

in cache has to be evicted, in order to make room for the new one. When a page fault occurs, the cost of the access operation is 1. Each page has to be served before the next request is presented. The goal is minimizing the total cost, i.e. the number of page faults. We use  $|\sigma|$  to denote the length of  $\sigma$ .

Given a sequence  $\sigma$ , the *cache state*  $\mathcal{C}(\sigma)$  is the set of pages in cache after  $\sigma$  has been served. The output of a paging algorithm corresponding to request sequence  $\sigma$  is the sequence  $\mathcal{C}(\sigma_1), \dots, \mathcal{C}(\sigma_{|\sigma|})$  of cache states, with  $\sigma_1 = \sigma(1)$  and  $\sigma_i = \sigma_{i-1}\sigma(i)$  for  $i > 1$ .

**Paging algorithms.** The core of a paging algorithm is the strategy according to which pages are evicted in the presence of faults. In this paper we analyze the performance of two well known heuristics for page eviction, namely LRU (Least Recently Used) and FWF (Flush When Full). The former is the best known algorithm for virtual memory management in operating systems [16] and has found application in web caching as well. The latter has mostly theoretical interest. For the sake of the analysis we are also interested in the off-line optimum. A well known optimal algorithm for paging is LFD (Longest Forward Distance). These algorithms are described below.

- **Longest Forward Distance (LFD):** Replace the page whose next request is latest. LFD is an optimal off-line algorithm and was proposed in an early work by Belady [2].
- **Least Recently Used (LRU):** When a page fault occurs, replace the page whose most recent request was earliest.
- **Flush When Full (FWF):** Assume the cache is initially empty. Three cases may occur by a request: i) the page requested is in cache: in this case the request is served at no extra cost; ii) a page fault occurs and the cache is not full: in this case the page is brought into cache, no page is evicted; iii) a page fault occurs and the cache is full: in this case, *all* pages in the cache are evicted, i.e. the cache is flushed. Note that both in case ii) and iii) the cost incurred by the algorithm is 1.

Both LRU and FWF belong to the class of *marking algorithms*. Roughly speaking, marking algorithms try to keep recently requested pages in cache, under the assumption that pages that have been requested in the recent past are more likely to be requested in the near future. Many marking algorithms have been proposed in practice, the most prominent ones are described in [3,16].

**Contribution of this paper.** Theory shows that LRU and the naive FWF heuristic have the same competitive ratio, practice provides pretty different results [18]: LRU’s performance is almost optimal on most instances of practical interest, whereas FWF often behaves poorly, as intuition suggests. This discrepancy is a consequence of the worst case approach followed by competitive analysis, which fails to capture the phenomenon of locality of reference [16].

In this paper, we extend the *diffuse adversary* approach introduced by Koutsopoulos and Papadimitriou [13]. In their model, the adversary is constrained to generate the input sequence according to a given family of probability distributions. We extend the work in [13] and [22,20] in the following way: i) We

propose a new framework in which the probability that some page  $p$  is requested, conditioned to any request sequence prior to  $p$ , also depends on how recently  $p$  was last requested; ii) We prove that, under a very large class of distributions, LRU's competitive ratio rapidly tends to become constant as locality increases, as observed in practice; iii) We assess the poor behaviour of FWF, by proposing a family of distributions parametrized by locality of reference and proving that FWF's performance degrades as locality increases. Together with the former ones, this result sharply discriminates between the behaviours of LRU and FWF, as observed in practice.

**Related results.** Sleator and Tarjan [15] introduced competitive analysis and were the first to show that FIFO and LRU are  $k$ -competitive. They also proved that this is the lower bound for any paging algorithm. Successively, Young and Torng [21,17] proved that all paging algorithms falling in two broad classes, including FIFO, LRU and FWF, are  $k$ -competitive. Torng [17] also proved that all marking algorithms are  $\Theta(k)$ -competitive<sup>1</sup>.

A considerable amount of research was devoted to overcoming the apparent discrepancies between theory and practice. Due to lack of space, we give a non-chronological overview of research in the field, giving emphasis to contributions that are closer to our approach in spirit.

The results of [13] were extended by Young [22,20], who proved that the competitive ratio of a subclass of marking algorithms (not including FWF) rapidly becomes constant as  $\epsilon = O(1/k)$ , is  $\Theta(\log k)$  when  $\epsilon = 1/k$  and rapidly becomes  $\Omega(k)$  as  $\epsilon = \Omega(1/k)$ . He also proved that, under this adversary, this is asymptotically the lower bound for any deterministic on-line algorithm. The role of randomization was explored by Fiat et al. [9], who proposed a randomized paging algorithm MARK that is  $H_k$ -competitive, a result which is tight.

Borodin et al. [4] assume that pages are vertices of an underlying graph, called *access graph*. Locality is modelled by constraining request sequences to correspond to simple walks in the access graph. The same model was used and extended in subsequent works [11,10,14,4,14] and it was extended in [8] to multi-pointer walks in the access graph.

The work of [12] is similar in spirit, although the approach is probabilistic: the authors assume that the request sequence is generated according to a Markov chain  $M$ . They consider fault rate, i.e. the long term frequency of page faults with respect to  $M$ .

A deterministic approach based on the working set of Denning [7] is proposed by Albers et al. [1].

Young [18,19] introduces the notion of loose competitiveness. A paging algorithm  $A$  is loosely  $c(k)$ -competitive if, for any request sequence, for all but a small fraction of the possible values for  $k$ , the cost of the algorithm is within  $c(k)$  times the optimum for every request sequence. The author proves [18] that a broad class of algorithms, including marking algorithms, is loosely  $\Theta(\log k)$ -competitive.

---

<sup>1</sup> Torng's result is actually more general, but this is the aspect that is relevant to the topics discussed in this paper.

Finally, Boyar et al. [6] analyze the relative worst order ratios of some prominent paging algorithms, including LRU, FWF and MARK. The worst order ratio of two algorithms  $A$  and  $B$ , initially proposed for bin packing [5] is defined as the ratio of their worst case costs, i.e. their worst case number of faults in the case of paging, over request sequences of the same length.

**Roadmap.** This paper is organized as follows: In Section 2 we propose our model and discuss some preliminaries. In Section 3 we provide a diffuse adversary analysis of LRU, where the distributions allowed to the adversary are used to model temporal locality. We prove that LRU is constant competitive with respect to a wide class of distributions. In Section 4 we prove that, under the same diffuse adversary, the performance of FWF is extremely poor. For the sake of brevity and readability, many proofs are omitted and will be presented in the full version of the paper.

## 2 Model and Preliminaries

The *diffuse adversary* model was proposed by Koutsoupias and Papadimitriou [13]. It removes the overly pessimistic assumption that we know nothing about the distribution according to which the input is generated, instead assuming that it is member of some known family  $\Delta$ . We say that algorithm  $A$  is  $c$ -competitive against the  $\Delta$ -diffuse adversary if, for every distribution  $D \in \Delta$  over the set of possible input sequences,

$$\mathbf{E}_D[A(\sigma)] \leq c\mathbf{E}_D[\text{OPT}(\sigma)] + b,$$

where  $\sigma$  is generated according to  $D$  and  $b$  is a constant. The *competitive ratio* of  $A$  against the  $\Delta$ -diffuse adversary is the minimum  $c$  such that the condition above holds for every  $D \in \Delta$ . In the sequel, we drop  $D$  from the expression of the expectation when clear from context.

The request sequence is a stochastic process. Without loss of generality, in the following we consider request sequences of length  $n$ . Considered any request sequence  $\sigma$  and its  $i$ -th request  $\sigma(i)$ , we call  $\sigma(1) \cdots \sigma(i-1)$  and  $\sigma(i+1) \cdots \sigma(n)$  respectively the *prefix* and the *postfix* of  $\sigma(i)$ . We define the following random variables: If  $i \leq n$ ,  $\Gamma(i)$  denotes the  $i$ -th page requested. If  $i \leq n$ ,  $\bar{\Gamma}(i)$  denotes the prefix of the  $i$ -th request.

### 2.1 Distribution of Page References

Similarly to [13] and [22], the family of distributions we consider is completely defined by the probability  $\mathbf{P}[p|\sigma]$  that  $p$  is requested provided  $\sigma$  is the prefix, for every  $p \in \mathcal{P}$  and for every possible prefix  $\sigma$  that satisfies  $|\sigma| \leq n-1$ . In particular, we assume that  $\mathbf{P}[p|\sigma]$  belongs to some family  $\Delta$ . The following definition will be useful to define some of them.

**Definition 1.** Consider a monotone, non increasing function  $f : I \rightarrow [0, 1]$ , where  $I = \{1, \dots, N\}$  for some positive integer  $N$ . We say that  $f$  is **concave** in

*I if, for all  $i = 2, \dots, N-1$ :  $f(i-1) - f(i) \leq f(i) - f(i+1)$ .  $f$  is **convex** in*  
*I if, for all  $i = 2, \dots, N-1$ :  $f(i-1) - f(i) \geq f(i) - f(i+1)$ .*

Assume  $\sigma$  is the request sequence observed so far. Provided both pages  $p_{j_1}$  and  $p_{j_2}$  appear in request sequence  $\sigma$ , we say that  $p_{j_1}$  is *younger* than  $p_{j_2}$  if the last request for  $p_{j_1}$  in  $\sigma$  occurs after the last request for  $p_{j_2}$ , while we say that  $p_{j_1}$  is *older* than  $p_{j_2}$  otherwise. The age of any page  $p_j$  with respect to  $\sigma$  is then defined as follows:  $\mathbf{age}(p_j, \sigma) = l$ , if  $p_j$  is the  $l$ -th most recently accessed page, while  $\mathbf{age}(p_j, \sigma) = \infty$  if  $p_j$  does not appear in  $\sigma$ . For the sake of simplicity, we assume in the sequel that  $\mathbf{age}(p_j, \sigma) < \infty$  for every  $p_j$ . This assumption can be removed by slightly modifying the definitions that follow. This is shown in the full version of the paper, for brevity. We now define, for every prefix sequence  $\sigma$ , a total order  $\mathcal{F}(\sigma)$  on  $\mathcal{P}$  by ordering pages according to their relative ages. More precisely, for any pages  $p_{j_1}, p_{j_2}$ ,  $p_{j_1} \prec_{\mathcal{F}(\sigma)} p_{j_2}$  if and only if  $\mathbf{age}(p_{j_1}, \sigma) < \mathbf{age}(p_{j_2}, \sigma)$ .  $\mathcal{F}(\sigma)$  obviously implies a one-to-one correspondence  $f_\sigma : \mathcal{P} \rightarrow \{1, \dots, M\}$ , with  $f_\sigma(p_j) = l$  if, given  $\sigma$ ,  $p_j$  is the  $l$ -th most recently accessed page.  $\mathbf{P}[p | \sigma]$  in turn may be regarded as a distribution over  $\{1, \dots, M\}$  by defining  $\bar{\mathbf{P}}[x | \sigma] = \mathbf{P}[f_\sigma^{-1}(x) | \sigma]$ , for every  $x = 1, \dots, M$ . So, for instance, by saying that  $\mathbf{P}[p | \sigma]$  has average value  $\mu$ , we mean that  $\bar{\mathbf{E}}[x | \sigma] = \sum_{x=1}^M x \bar{\mathbf{P}}[x | \sigma] = \mu$ .

## 2.2 Diffuse Adversaries

We next define three families of probability distributions according to which, given any prefix  $\sigma$ , the next page request can be generated.

**Definition 2.**  $\mathcal{G}(\tau)$  is the family of probability distributions over  $\{1, \dots, M\}$  that have expected value at most  $k/2$  and standard deviation  $\tau$ .  $\mathcal{D}_1$  (resp.  $\mathcal{D}_2$ ) are the families of distributions that are monotone, non-increasing and concave (resp. convex) in  $\{1, \dots, M\}$ .

The first property of Definition 2 models the well known folklore that “90 percent of requests is for about 10 percent of the pages”.

Figure 1 shows an example in which  $\mathbf{P}[p | \sigma] \in \mathcal{D}_2$ . The  $y$  coordinate is  $\bar{\mathbf{P}}[x | \sigma]$ . Observe that the  $l$ -th element along the  $x$ -axis represents the  $l$ -th most recently requested page and thus the first  $k$  elements correspond to the set of recent pages. In the case of LRU these are also the pages in the cache of the algorithm.

We next describe how the adversary generates request sequences. Assumed a prefix request sequence  $\sigma$  of length  $i < n$  has been generated, the adversary picks a distribution  $D(i)$  out of some family  $\Delta$  and it chooses the next page to request ( $\sigma(i+1)$ ) according to  $D(i)$ . Observe that, in general,  $D(i_1) \neq D(i_2)$  for  $i_1 \neq i_2$ . We define different adversaries, according to how  $D(i)$  is chosen:

**Definition 3.** An adversary is concentrated (respectively concave and convex) if  $D(i) \in \mathcal{G}(\tau)$  (respectively  $\mathcal{D}_1$  and  $\mathcal{D}_2$ ) for all  $i$ . If  $D(i) = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \mathcal{G}(\tau)$  for some  $\tau$  and for all  $i$  we speak of a general adversary.

Observe that the general adversary resumes all possible subcases.

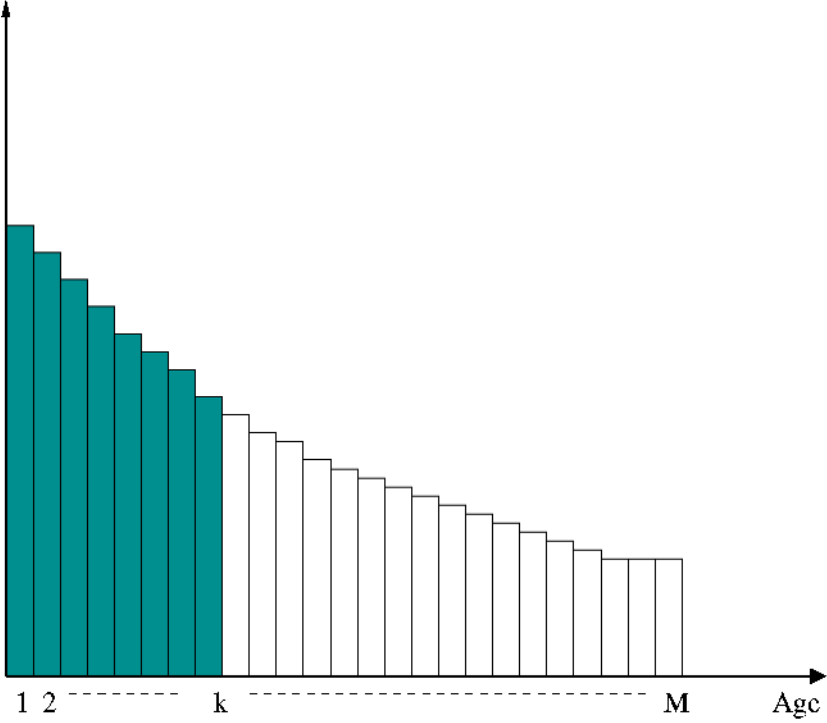


Fig. 1. This picture represents the case in which  $\mathbf{P}[p | \sigma] \in \mathcal{D}_2$ .

### 2.3 Preliminary Results

We partition each request sequence into *phases* in the usual way [3]. In particular, phases are defined inductively as follows: i) phase 1 starts with the first request and completes right before the  $(k + 1)$ -th *distinct* request has been presented; ii) for  $i > 1$ , assuming phase  $i - 1$ , phase  $i$  starts with the  $(k + 1)$ -th distinct request after the start of phase  $i - 1$  and it completes right before the  $(2k + 1)$ -th distinct request after the start of phase  $i - 1$ . We point out that the last phase of a request sequence might contain less than  $k$  distinct requests. However, it has to contain at least one. In the sequel,  $F$  is the random variable that denotes the overall number of phases.

We say that a page is *marked* in a phase, if it was already requested during the phase. Otherwise the page is said *unmarked*. A *marking* algorithm never evicts a marked page when a page fault occurs. Observe that both LRU and FWF are marking algorithms. As such, they both achieve the best possible competitive ratio  $k$  for on-line paging algorithms [17].

Considered any phase  $\ell \geq 1$ , we say that a page  $p_j$  is *new* in  $\ell$  if it was not requested during phase  $\ell - 1$ . Otherwise we say that  $p_j$  is *old*. Also, considered any point of  $\ell$ , any of the  $k$  most recently requested pages is said *recent*. Considered any prefix  $\sigma$ , we denote by  $\mathcal{N}(\sigma)$  and  $\mathcal{R}(\sigma)$  respectively the set of new and recent



pages after request subsequence  $\sigma$  has been presented. Observe that both  $\mathcal{N}(\sigma)$  and  $\mathcal{R}(\sigma)$  do not depend on the paging algorithm. We say that page  $p_j$  is *critical*, if  $p_j$  is old but it is not currently in cache. Observe that, considered any prefix  $\sigma$ , the sets of in cache and critical pages depend on the particular paging algorithm under consideration. In the sequel, for any paging algorithm  $A$ , we denote by  $\mathcal{C}_A(\sigma)$  and  $\mathcal{O}_A(\sigma)$  respectively the sets of in cache and critical pages. Observe that  $\mathcal{C}_A(\sigma) \neq \mathcal{R}(\sigma)$  in general. Also, we write  $\mathcal{C}(\sigma)$  for  $\mathcal{C}_A(\sigma)$  and  $\mathcal{O}(\sigma)$  for  $\mathcal{O}_A(\sigma)$  whenever  $A$  is clear from context.

$W^\ell$  and  $N^\ell$  are random variables whose values are respectively the number of requests to critical and new pages during phase  $\ell$ .  $W^\ell = N^\ell = 0$  if no  $\ell$ -th phase exists. A request in phase  $\ell$  is said *distinct* if it specifies a page that was not previously requested during the phase. Note that requests for new pages are distinct by definition. We denote by  $D^\ell$  the number of distinct requests during phase  $\ell$ . Observe that, differently from  $N^\ell$  and  $D^\ell$ ,  $W^\ell$  depends on the paging algorithm. However, in the rest of this paper the algorithm is always understood from context, hence we do not specify it for these variables. Finally, considered any random variable  $V$  defined over all possible request sequences of length  $n$ , we denote by  $V_\sigma$  its deterministic value for a particular request sequence  $\sigma$ . The following result is due to Young [22]:

**Theorem 1 ([22]).** *For every request sequence  $\sigma$ :*

$$\frac{1}{2} \sum_{\ell} N_{\sigma}^{\ell} \leq OPT(\sigma) \leq \sum_{\ell} N_{\sigma}^{\ell}.$$

In the sequel, we assume that the first request of the first phase is for a new page. We can assume this without loss of generality, since a sequence of requests for recent pages does not change the cache state of LRU and has cost 0. This allows us to state the following fact:

**Fact 1** *The first request of every phase is for a new page.*

The following fact, whose proof is straightforward, gives a crucial property of LRU:

**Fact 2**  $\mathcal{C}_{LRU}(\sigma) = \mathcal{R}(\sigma)$ , *for every  $\sigma$ .*

Observe that Fact 2 in general does not hold for other (even marking) algorithms. Set  $\ell(n) = \lceil n/k \rceil$ . The following fact will be useful in the sequel:

**Fact 3**  $F \leq \ell(n)$  *deterministically for request sequences of length  $n$ .*

### 3 Analysis of LRU

The main goal of this section is to bound

$$\frac{\mathbf{E}[LRU(\sigma)]}{\mathbf{E}[OPT(\sigma)]},$$

when  $\sigma$  is generated by any of the adversaries defined above. We can express the competitive ratio of LRU as stated in the following lemma:

**Lemma 1.**

$$\frac{\mathbf{E}[LRU(\sigma)]}{\mathbf{E}[OPT(\sigma)]} \leq 2 \frac{\sum_{\ell=1}^{\ell(n)} \mathbf{E}[W^\ell | F \geq \ell] \mathbf{P}[F \geq \ell]}{\sum_{\ell=1}^{\ell(n)} \mathbf{E}[N^\ell | F \geq \ell] \mathbf{P}[F \geq \ell]} + 2. \quad (1)$$

We now introduce the following random variables, for all pairs  $(\ell, j)$ , where  $j = 1, \dots, k$ : i)  $X_j^\ell = 1$  if there is an  $\ell$ -th phase, there is a distinct  $j$ -th request in the  $\ell$ -th phase and this request is for a critical page,  $X_j^\ell = 0$  otherwise (i.e. no  $\ell$ -th phase exists or the phase is not complete and it does not contain a  $j$ -th distinct request or it does, but this request is not for a critical page); ii)  $Y_j^\ell = 1$  if the  $j$ -th distinct request in phase  $\ell$  is to a new page,  $Y_j^\ell = 0$  otherwise; a consequence of the definitions above is that we have  $\sum_{j=1}^k X_j^\ell = W^\ell$  and  $\sum_{j=1}^k Y_j^\ell = N^\ell$ . Observe, again, that  $Y_j^\ell$  does not depend on the paging algorithm considered, while  $X_j^\ell$  does. Again, the algorithm is always understood from context.

The rest of our proof goes along the following lines: intuitively, in the general case we have that, as  $\bar{\mathbf{P}}[x | \sigma]$  becomes concentrated (i.e. variance decreases), most requests in the phase are for pages in cache, with the exception of a small number that at some point rapidly decreases with variance. In the other cases, we are able to bound the expected number of requests for old pages with the expected number of requests for new pages. We point out that in all cases the proof crucially relies on Fact 2.

### 3.1 Convex Adversaries

We consider the case in which  $D(i) \in \mathcal{D}_2$ , for every  $i = 1, \dots, n$ . Intuitively, we shall prove that if the probability of requesting a critical page is more than twice the probability of requesting a new page, then the former has to be sufficiently small in absolute terms. For the rest of this subsection,  $N_j^\ell$  is a random variable that denotes the number of requests for new pages that are presented before the  $j$ -th distinct request of phase  $\ell$ . In particular,  $N_j^\ell = 0$  if no  $\ell$ -th phase exists while  $N_j^\ell = N^\ell$  if the  $\ell$ -th phase is not complete and it does not contain a  $j$ -th distinct request. We now state the following lemma:

**Lemma 2.** *If the request sequence is generated by a convex adversary and  $M \geq 4k$ ,*

$$\mathbf{P}[X_j^\ell = 1 | F \geq \ell] \leq 2\mathbf{P}[Y_j^\ell = 1 | F \geq \ell] + \frac{16(j-1)}{k^2} \mathbf{E}[N^\ell | F \geq \ell].$$

*Sketch of proof.* Observe that the claim trivially holds if  $j = 1$ , since no critical pages exist in a phase before the first new page is requested. For  $j \geq 2$  we can write:

$$\mathbf{P}[X_j^\ell = 1 | F \geq \ell] = \sum_{x=1}^{j-1} \mathbf{P}[X_j^\ell = 1 | F \geq \ell \cap N_j^\ell = x] \mathbf{P}[N_j^\ell = x | F \geq \ell].$$

The assumption of convexity implies that, if  $\mathbf{P}[X_j^\ell = 1 \mid F \geq \ell \cap N_j^\ell = x] > 2\mathbf{P}[Y_j^\ell = 1 \mid F \geq \ell \cap N_j^\ell = x]$ , then the former is bounded by  $16x^2/k^2$ . As a consequence:

$$\mathbf{P}[X_j^\ell = 1 \mid F \geq \ell \cap N_j^\ell = x] \leq 2\mathbf{P}[Y_j^\ell = 1 \mid F \geq \ell \cap N_j^\ell = x] + \frac{16x^2}{k^2}.$$

We continue with:

$$\begin{aligned} \mathbf{P}[X_j^\ell = 1 \mid F \geq \ell] &= \sum_{x=1}^{j-1} \mathbf{P}[X_j^\ell = 1 \mid F \geq \ell \cap N_j^\ell = x] \mathbf{P}[N_j^\ell = x \mid F \geq \ell] \\ &\leq 2\mathbf{P}[Y_j^\ell = 1 \mid F \geq \ell] + \frac{16(j-1)}{k^2} \sum_{x=1}^{j-1} x \mathbf{P}[N_j^\ell = x \mid F \geq \ell] \\ &\leq 2\mathbf{P}[Y_j^\ell = 1 \mid F \geq \ell] + \frac{16(j-1)}{k^2} \mathbf{E}[N^\ell \mid F \geq \ell], \end{aligned}$$

where the second inequality follows since  $x \leq j-1$ , while the third inequality follows since  $N_j^\ell \leq N^\ell$  by definition.

We can now write:

$$\mathbf{E}[W^\ell \mid F \geq \ell] \leq 2\mathbf{E}[N^\ell \mid F \geq \ell] + \frac{16}{k^2} \mathbf{E}[N^\ell \mid F \geq \ell] \sum_{j=1}^k (j-1) < 10\mathbf{E}[N^\ell \mid F \geq \ell],$$

where the second inequality follows from simple manipulations. Recalling Equation (1) we can finally state the following Theorem:

**Theorem 2.** *If the request sequence is generated by a convex adversary,*

$$\frac{\mathbf{E}[LRU(\sigma)]}{\mathbf{E}[OPT(\sigma)]} \leq 22.$$

Note that Theorem 2 does not necessarily hold for other marking algorithms, since the proof of Lemma 2 crucially uses Fact 2.

### 3.2 Concentrated Adversary

In this subsection we consider all adversaries such that  $D(i) \in \mathcal{G}(\tau)$  for every  $i = 1, \dots, n$ . We can state the following lemma:

**Lemma 3.** *If the request sequence is generated by a concentrated adversary then, for  $j \geq 2$ ,*

$$\mathbf{P}[X_j^\ell = 1 \cup Y_j^\ell = 1 \mid F \geq \ell] \leq \left( \frac{2\tau}{k} \right)^2.$$

Lemma 3 and Fact 1 imply  $\mathbf{E}[W^\ell | F \geq \ell] = \mathbf{E}\left[\sum_{j=1}^k X_j^\ell | F \geq \ell\right] \leq (k-1)(2\tau/k)^2$ . We can therefore conclude with:

**Theorem 3.** *If the request sequence is generated by a concentrated adversary,*

$$\frac{\mathbf{E}[LRU(\sigma)]}{\mathbf{E}[OPT(\sigma)]} \leq \frac{8\tau^2}{k} + 2.$$

### 3.3 Concave Adversaries

In this section we provide an analysis of LRU for the case of concave adversaries. We start with the following Lemma:

**Lemma 4.** *If the request sequence is generated by a concave adversary and  $M \geq 4k$ ,*

$$\mathbf{P}[X_j^\ell = 1 | F \geq \ell] \leq 2\mathbf{P}[Y_j^\ell = 1 | F \geq \ell].$$

**Corollary 1.** *If the request sequence is generated by a concave adversary and  $M \geq 4k$ ,*

$$\frac{\mathbf{E}[W^\ell | F \geq \ell]}{\mathbf{E}[N^\ell | F \geq \ell]} \leq 2.$$

Corollary 1 and Equation (1) immediately imply the following Theorem:

**Theorem 4.** *If the request sequence is generated by a concave adversary and  $M \geq 4k$ ,*

$$\frac{\mathbf{E}[LRU(\sigma)]}{\mathbf{E}[OPT(\sigma)]} \leq 6.$$

Theorem 4 does not necessarily hold for other marking algorithms, since its proof crucially uses Fact 2. Observe, also, the interesting circumstance that this adversary includes the uniform distribution as a special case. For this particular distribution, the results of Young [20,22] imply a competitive ratio  $\Omega(1)$  as  $M = k + o(k)$ , that rapidly becomes  $\Omega(\log k)$  as  $M = k + O(1)$ .

### 3.4 General Adversaries

If  $D(i) \in \mathcal{G}(\tau) \cup \mathcal{D}_1 \cup \mathcal{D}_2$  for every  $i$ , the following is a corollary of the results of Subsections 3.1, 3.2, and 3.3:

**Theorem 5.** *If the request sequence is generated by a general adversary,*

$$\frac{\mathbf{E}[LRU(\sigma)]}{\mathbf{E}[OPT(\sigma)]} \leq \max \left\{ 22, \frac{8\tau^2}{k} + 2 \right\}.$$

## 4 A Lower Bound for FWF

We prove that under very general distributions, the performance of FWF degrades to  $\Omega(k)$ -competitiveness as locality increases, while the opposite occurs for LRU. This behaviour is in line with practical experience (see e.g. [18,3]). Together with the results of Section 3, this seems to us one theoretical validation of the great difference observed between the behaviour of FWF and other marking algorithms (in particular LRU). We prove the following

**Theorem 6.** *Assume  $0 < \alpha < 1$  and assume request sequences of length  $n \geq 2(k+1)/\alpha$  are generated by a diffuse adversary that, for every  $i \leq n$  and for every  $\sigma$  such that  $|\sigma| = i - 1$ , satisfies the following constraint*

$$\mathbf{P}[\Gamma(i) \in \mathcal{R}(\sigma) \mid \bar{\Gamma}(\sigma) = \sigma] = 1 - \alpha.$$

*Then,*

$$\frac{\mathbf{E}[FWF(\sigma)]}{\mathbf{E}[OPT(\sigma)]} > \frac{k}{2((k-1)\alpha + 1)}.$$

$$\frac{\mathbf{E}[LRU(\sigma)]}{\mathbf{E}[OPT(\sigma)]} \leq 2\alpha k + 2.$$

Observe that  $\alpha$  measures the amount of locality, i.e. locality increases as  $\alpha$  becomes smaller. This theorem thus states that the long term competitive ratio of FWF can be as bad as  $\Omega(k)$  as locality increases, whereas under the same circumstances the competitive ratio of LRU tends to become constant. As an example, in the case of a general adversary Theorem 5 implies that this occurs as the standard deviation becomes  $O(\sqrt{k})$ .

**Acknowledgments.** The author wishes to thank his colleagues and friends Stefano Leonardi and Alberto Marchetti-Spaccamela for their suggestions and support.

## References

1. S. Albers, L. M. Favrholdt, and O. Giel. On paging with locality of reference. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC-02)*, pages 258–267, 2002.
2. L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
3. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
4. A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 249–259, 1991.

5. J. Boyar, L. M. Favrholdt, and K. S. Larsen. The relative worst case order ratio applied to paging. In *Proceedings of the 5th Italian Conference on Algorithms and Complexity*, pages 58–69, 2003.
6. J. Boyar, L. M. Favrholdt, and K. S. Larsen. The relative worst order ratio applied to paging. Tech. report ALCOMFT-TR-03-32, Future and Emerging Technologies program under the EU, contract number IST-1999-14186, 2003.
7. P. J. Denning. The working set model for program behaviour. *Communications of the ACM*, 11(5):323–333, 1968.
8. A. Fiat and A. R. Karlin. Randomized and multipointer paging with locality of reference. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 626–634, 1995.
9. A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.
10. A. Fiat and M. Mendel. Truly online paging with locality of reference. In *38th IEEE Annual Symposium on Foundations of Computer Science*, pages 326–335, 1997.
11. S. Irani, A. R. Karlin, and S. Phillips. Strongly competitive algorithms for paging with locality of reference. *SIAM Journal on Computing*, 25(3):477–497, 1996.
12. A. R. Karlin, S. J. Phillips, and P. Raghavan. Markov paging. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 208–217, 1992.
13. E. Koutsoupias and C. H. Papadimitriou. Beyond competitive analysis. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 394–400, 1994.
14. M. Chrobak and J. Noga. LRU is better than FIFO. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-98)*, pages 78–81, 1998.
15. D. Sleator and R. E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28:202–208, 1985.
16. Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.
17. E. Torng. A unified analysis of paging and caching. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 194–203, 1995.
18. N. Young. The  $k$ -server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.
19. N. E. Young. On-line file caching. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-98)*, pages 82–86, 1998.
20. N. E. Young. On-line paging against adversarially biased random inputs. *Journal of Algorithms*, 37(1):218–235, 2000.
21. Neal Young. *Competitive paging and dual-guided on-line weighted caching and matching algorithms*. PhD thesis, Department of Computer Science, Princeton University, 1991.
22. Neal E. Young. Bounding the diffuse adversary. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 420–425, 1998.

# An Algorithm for Computing DNA Walks<sup>\*</sup>

Ankur Bhargava and S. Rao Kosaraju

Department of Computer Science  
Johns Hopkins University  
Baltimore, MD 21218.

**Abstract.** The process of reading a DNA molecule can be divided into three phases: *sequencing*, *assembly* and *finishing*. The draft sequence after assembly has gaps with no coverage and regions of poor coverage. Finishing by walks is a process by which the DNA in these regions is resequenced. Selective sequencing is expensive and time consuming. Therefore, the laboratory process of finishing is modeled as an optimization problem aimed at minimizing laboratory cost. We give an algorithm that solves this problem optimally and runs in worst case  $O(n^{1\frac{3}{4}})$  time.

## 1 Introduction

The aggregate DNA in the human genome is about 3 billion base pairs long. In the course of sequencing, a DNA sample is replicated a number of times and clipped by restriction enzymes into smaller DNA molecules. The position at which a DNA molecule is cut depends on the restriction enzyme used and the target site to which it binds. These shorter DNA molecules are fed into sequencers which read off about 500 base pairs from one side of each molecule. Each 500 base pair length sequence read by a sequencer is called a *read*. In shotgun sequencing, the sequencer can read from both ends. The more the reads the lesser are the unsequenced portions of the genome. Typically, a laboratory will resort to 6 fold coverage. This means that in this example the sequencers will read a total of about 18 billion base pairs. If we assume that each *read* of 500 base pairs is selected uniformly and independently at random from the DNA, then on average, 6% of the DNA is read less than 3 times. These regions of DNA are considered to be poorly covered. Sequencing machines are not a hundred percent accurate and occasionally will misread a base, skip a base or insert a nonexistent base. In order to guarantee some level of accuracy, it is common to demand that each base pair in the sequenced DNA be read some number of times. This gives some assurance as to the quality of data. The laboratory has to selectively resequence those parts of the genome which have poor coverage. Selectively targeting parts of DNA is an expensive and time consuming process. The selected region that must be re-read, is identified by a primer in the neighborhood of the target. This region is then amplified and re-read. This process of targeted sequencing aimed at improving coverage is called

---

<sup>\*</sup> Supported by NSF Grant CCR-0311321.

*finishing*. The bottom line is that the more the coverage the less the likelihood of error. There are various ways to benchmark the quality of sequenced DNA. One such measure is called “Bermuda Quality” [Hgp98]. Bermuda quality addresses the fact that DNA is double stranded. The two strands of DNA are referred to as the *Watson* and the *Crick* strands. The two strands are complements of each other. Essentially, Bermuda quality requires that every base pair must be read at least twice on one strand and at least once on the complementary strand. There is a tradeoff between cheap random sequencing versus targeted sequencing [CKM<sup>+</sup>00]. Random sequencing has diminishing returns and each targeted resequencing has a high unit cost. The tradeoff is made by selecting a coverage ratio, such as the 6 fold coverage in the example above.

Constructing the primer is the most expensive step in such a laboratory procedure. Resequenced reads are called *walks* as opposed to the untargeted ones which are called *reads*. After sequencing and assembly, we are left with regions of DNA which do not meet the Bermuda quality requirement. The laboratory must add walks to *finish* the sequence. Each walk requires the construction of a primer. Once a primer is constructed for some site in the genome, it may be reused for a second walk at the same site on the same strand. This second walk has a lower laboratory cost because the same primer is reused. This was introduced by [CKM<sup>+</sup>00] in modeling this problem.

Naturally this problem has been posed as an optimization problem. The goal is to minimize the cost of resequencing needed. Programs that perform such optimization are described in [GAG98,GDG01]. They do not bound running times. This problem’s complexity was first studied by [PT99] who formulated it as an integer linear program. The cost model we study in this paper was shown to have an  $O(n^4)$  dynamic programming solution in [CKM<sup>+</sup>00]. Hart [Har02] gave the first sub-cubic algorithm for this problem. [Har02] also discusses a randomized model in which their algorithm runs in expected linear time.

We develop an algorithm that has a worst case running time of  $O(n^{1\frac{3}{4}})$ . Our algorithm also guarantees the expected  $O(n)$  running time for the randomized model discussed in [Har02] without any modification. The problem is introduced in section 2 and the basic algorithm is developed in section 3. Further improvements are discussed in section 4.

## 2 Model and Problem

Let the *problem interval* be  $[0, n) \subset \mathbb{R}$  where  $n$  is any real number greater than 1. A *read* is a unit length interval on the real number line. Each read,  $r$ , has a start position,  $p(r) \in [0, n - 1)$ , a direction,  $d(r) \in \{\leftarrow, \rightarrow\}$  and is associated with the unit real interval  $[p(r), p(r) + 1)$ . Given a set of reads,  $S$ , the goal is to find a set of unit intervals called *walks*,  $W$ . Each walk,  $w \in W$ , has a position,  $p(w) \in [0, n)$ , a direction,  $d(w) \in \{\leftarrow, \rightarrow, \Leftarrow, \Rightarrow\}$  and is associated with the unit real interval  $[p(w), p(w) + 1)$ .

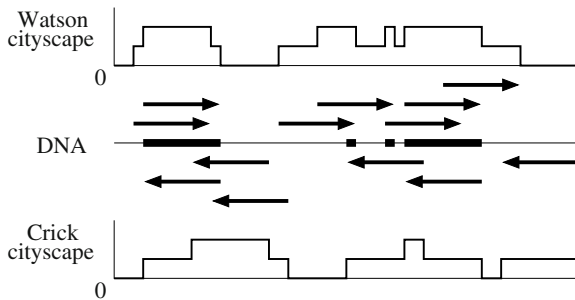
Any point in the problem interval is *covered* by a walk or read if it lies inside the interval corresponding to the walk or read. A point covered by a walk with



direction  $\leftarrow$  is the same as if it were covered by a read with direction  $\leftarrow$ . Similarly a walk with direction  $\rightarrow$  is the same as a read with direction  $\rightarrow$ . A point covered by a walk with direction  $\Leftarrow$  is the same as if it were covered by two reads of direction  $\leftarrow$ . Similarly a walk with direction  $\Rightarrow$  is same as two reads with direction  $\rightarrow$ . A point is *adequately covered* by the reads and walks if and only if it lies inside at least 2 unit intervals with direction  $\rightarrow$  and at least 1 unit interval with direction  $\leftarrow$  or it lies inside at least 2 intervals with direction  $\leftarrow$  and 1 interval with direction  $\rightarrow$ . Each walk has an associated cost, denoted:  $c(\cdot)$ .  $c(\rightarrow) = c(\leftarrow) = 1$  and  $c(\Leftarrow) = c(\Rightarrow) = 1 + \epsilon$ , where  $0 < \epsilon < 1$  and  $\epsilon$  is a rational number. The cost of a set of walks,  $W$ , is equal to the sum of the costs of the walks in  $W$ . In addition, a set of reads,  $S$ , and a set of walks,  $W$ , *adequately cover* the problem interval iff every point in the problem interval is adequately covered.

In the above formulation the use of the real line is just a convenience and can easily be replaced with integers. The set  $S$  corresponds to the set of reads obtained from the first round of sequencing. For simplicity, each read is assumed to be of unit length. The set of walks in the solution correspond to the required resequencing. The interval is adequately covered when the union of reads and walks meet Bermuda quality. Walks of the type  $\Leftarrow$  and  $\Rightarrow$  are the equivalent of performing two walks on the same strand and at the same site. This is reflected in the lab cost of performing such walks:  $1 + \epsilon$ . Two  $\leftarrow$ 's cost 2 while one  $\Leftarrow$  costs less than 2.

**Finishing:** Given a problem instance,  $S$ , find a set of walks  $W$  with minimum cost such that  $[0, n]$  is adequately covered.



**Fig. 1.** The two “cityscapes”.

Figure 1 is an example of a problem instance. Each arrow denotes a unit length interval. Some reads are on the Watson strand and some are on the Crick strand. The bold parts of the center line highlight the regions of DNA that are adequately covered (of Bermuda quality) by the given reads. Czabarka et al gave the term *cityscape* to describe the coverage on one strand as a function that takes

values 0, 1 or 2. The cityscape above (resp. below) the center line specifies the coverage on the Watson strand (resp. Crick strand). If a base on a particular strand is read more than twice, then this gives us no new information. It suffices to say that the cityscape is at a value of 2 at such a point. The coverage at a point is adequate when the sum of the two cityscapes is at least 3.

### 3 Algorithm Development

We start by introducing some definitions and notation. There may be many different solutions with the same cost. If we are given any optimum solution, we can shift each walk to the right for as long as this shifting does not violate the coverage requirement. Since the set of walks is the same, the solution remains at optimum. This is called the *shifted optimum*.

A *feasible partial solution* at  $x$ , is a set of walks,  $W$ , such that its walks are shifted, and every point in the interval  $[0, x)$  has adequate coverage and some point in  $[x, x + 1)$  has inadequate coverage. The set  $\mathcal{W}_x$  is the set of all feasible partial solutions at  $x$ . A *chain* in a *feasible partial solution* is a maximal sequence of walks  $(w_1, w_2, \dots, w_k)$  stacked end to end, such that  $p(w_{i+1}) = p(w_i) + 1$ , for  $i = 1, \dots, k-1$ . Keep in mind that a walk could be a part of more than one chain.

Figure 2 shows an optimum set of walks for the problem instance described in figure 1. Note that with the walks the sequence meets Bermuda quality. Figure 2 also shows two non-trivial chains and illustrates how two walks at the same point and the same strand reduce the overall cost. The cost of the walks in figure 2 is  $7 + 4\epsilon$ .

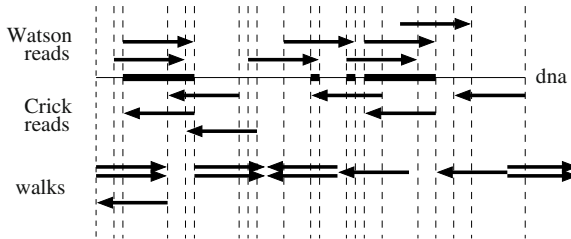


Fig. 2. A shifted optimum.

#### 3.1 The Problem Instance

A problem instance,  $S$ , is a set of reads in a problem interval of length  $n$ . The reads may be concentrated heavily in some part of the problem interval while there may be other parts that have no reads at all. A problem in which the read distribution varies wildly can be reduced to a new problem instance with a relatively even distribution of reads (with a bound on the read density) while

guaranteeing that the solution for the reduced instance can be used to recover a solution for the original problem in linear time.

**Lemma 1.** *Given any problem instance, it can be reduced to one in which:*

- (i) *there are no more than 8 reads that start<sup>1</sup> in any unit interval, and*
- (ii) *there is no stretch of length greater than 2 without a read.*

*Proof (i).* Assume that some unit interval has more than 4 reads that start within it and have the same direction. According to the requirements for adequate coverage there need be only 2 reads with like direction over any given point. So all reads that start after the first two but before the last two can be dropped. Similarly there are no more than 4 reads in the other direction.  $\square$

*Proof (ii).* Assume that there is a read-less stretch,  $(p(r_1) + 1, p(r_2))$ , such that  $p(r_2) - p(r_1) > 3$ , for  $r_1, r_2 \in S$ . Consider the shifted optimum solution  $W_{OPT}$  for this instance. Let  $B$  be the set of walks that start in the interval  $I_1 = [p(r_1) + 1, p(r_1) + 2)$ . All other walks in this interval must belong to the same chain as one of the walks in  $B$  (if not then  $W_{OPT}$  is not shifted). Since  $B$  adequately covers some point in  $I_1$  we can be assured of the fact that until interval  $I_2 = [p(r_1) + \lceil p(r_2) - p(r_1) \rceil - 2, p(r_1) + \lceil p(r_2) - p(r_1) \rceil - 1)$ , each interval is covered by a set of walks that costs no more than  $c(B)$  (if not then this is not OPT). Note that  $c(B)$  is the lowest possible cost required to cover a unit length read-less interval. So we can remove the interval  $[p(r_1) + 1, p(r_1) + \lceil p(r_2) - p(r_1) \rceil - 1)$  from the solution, there by creating a new problem instance without the long read-less stretch and a cost of  $c(W_{OPT}) - (\lceil p(r_2) - p(r_1) \rceil - 2) \cdot c(B)$ .

Assume that this solution is not the optimum for this instance, and let some  $W'$  is the optimum for this problem instance. Now using the same construction as above, but done in reverse, find  $W''$  which is a solution set to instance  $S$ . Note that  $c(W'') = c(W') + (\lceil p(r_2) - p(r_1) \rceil - 2) \cdot c(B)$ . This will mean that  $c(W'') < c(W_{OPT})$ . Hence the lemma holds.  $\square$

From here on we will assume that every problem instance has been stripped of unnecessary reads and has no read-less stretches of length more than 2 in accordance with lemma 1.

### 3.2 Characterization of the Optimum Solution

**Lemma 2.** *For any given problem,  $S$ , the shifted optimum,  $W_{OPT}$ , satisfies the following properties:*

- (i)  *$W_{OPT}$  has at most 3 walks that start at positions within any unit interval,*
- (ii)  *$W_{OPT}$  has at most 3 walks that cross any point,*
- (iii) *Every chain in  $W_{OPT}$  starts at the end of a read or at 0, and*

---

<sup>1</sup> Any reference to the beginning or start of a read or walk refers to its position,  $p(\cdot)$ , the direction has no relevance to where it starts. For example, a read  $[a, a + 1)$  starts at  $a$  even if its direction is  $\leftarrow$ .

(iv) If in  $W_{OPT}$ , 3 walks end inside a read then one of the first two must be the last walk of its chain.

*Proof (i).* Consider some interval  $[x, x + 1)$ . Let  $w_1, w_2, \dots, w_k \in W_{OPT}$  such that  $x \leq p(w_1) \leq p(w_2) \leq \dots \leq p(w_k) < x + 1$ . Note that  $W_{OPT}$  is a shifted optimum, so no walk can be shifted right without compromising coverage. For the sake of contradiction let  $k > 3$ . If  $d(w_1) = d(w_2) = d(w_3)$  then  $w_3$  can be shifted out of this interval. If any of the first 3 walks differ in direction then the interval has adequate coverage and hence  $w_4$  can be shifted out of the interval.  $\square$

*Proof (ii).* Consider any point  $x \in [0, n]$ . Let  $w \in W_{OPT}$  be such that  $x \in [p(w), p(w) + 1)$  and  $w$  is the walk with the lowest such  $p(w)$ . From lemma 2(i) at most 3 walks can start in the interval  $[p(w), p(w) + 1)$ . Hence there are no more than 3 walks that cross  $x$ .  $\square$

*Proof (iii).* Let  $w$  be the first walk in the given chain. Since this is a shifted optimum, shifting  $w$  to the right would have resulted in a point with inadequate coverage. Provided this point is not at 0, there exists a point just before this that does have adequate coverage. This means that either a read or a walk terminated at this point. Since the chain is maximal by definition, this cannot be a walk. So it must be a read.  $\square$

*Proof (iv).* Let  $w_1, w_2$  and  $w_3$  be the three walks that end inside read  $r \in S_n$  in a shifted optimum in ascending order of position. Assume for the sake of contradiction that some  $w_4, w_5 \in W$  satisfy  $p(w_4) = p(w_1) + 1$  and  $p(w_5) = p(w_2) + 1$ .

If  $p(w_4) < p(w_5) < p(w_3) + 1$  then in the interval  $(p(w_4), p(w_5))$ , walks  $w_3, w_4$  and read  $r$ , together provide adequate coverage. If this were not so,  $w_4$  serves no purpose and can be shifted out of  $(p(w_4), p(w_5))$ . Meanwhile in interval  $(p(w_5), p(w_3) + 1)$ ,  $w_4, w_3$  and  $r$  must provide adequate coverage, implying that  $w_5$  should be shifted out.

If  $p(w_4) = p(w_5) < p(w_3) + 1$  then over the interval  $(p(w_4), p(w_3) + 1)$ , irrespective of what direction  $r$  and  $w_3$  have, either  $w_4$  or  $w_5$  must be shifted right.

If  $p(w_4) = p(w_5) = p(w_3) + 1$ . We know from lemma 2(i) that there cannot be a third walk that starts in this interval. Since  $w_1, w_2$  and  $w_3$  start at the same point, we can let  $w_3$  be first.  $\square$

### 3.3 A Partial Solution

Let  $L$  be the length of the longest chain in any of the feasible partial solutions. Note that  $L \leq n$ . The parameter  $L$  is useful in considering other models, such as the one in which reads in the problem instance are randomly distributed. In such cases the longest possible chain is of constant length with high probability.

**Lemma 3.** Let  $\mathcal{W}_x$  be the set of feasible partial solutions at  $x \in [0, n)$ . Then,  $\mathcal{W}_x$  satisfies the following properties:

- (i) if  $W_1, W_2 \in \mathcal{W}_x$  then  $|c(W_1) - c(W_2)| \leq 2 + \epsilon$ ,
- (ii) if  $\epsilon$  is a constant rational and  $\mathcal{W}_x$  is partitioned into sets of equal cost then there are  $O(1)$  such sets, and
- (iii)  $|\mathcal{W}_x| = O(L)$ .

*Proof (i).* Let  $P$  be a set of walks such that  $W_1 \cup P$  is an optimum solution for the given problem. Let the cost of the optimum solution be  $\text{OPT}$ . In addition let  $c(W_1) = c(W_2) + \Delta$ , where  $\Delta$  is a positive number. We know that  $W_2$  covers every point before  $x$  adequately and  $P$  covers every point after  $x + 1$  adequately. In addition, a  $\leftarrow$  and  $\rightarrow$  are all that are needed to cover  $[x, x + 1)$  adequately. Therefore,  $c(W_2) + 2 + \epsilon + c(P) \geq \text{OPT}$ . Substituting for  $c(W_2)$ , we get,  $c(W_1) + c(P) + 2 + \epsilon - \Delta \geq \text{OPT}$ . We know that  $c(W_1) + c(P) = \text{OPT}$ . This implies that  $\Delta \leq 2 + \epsilon$ .  $\square$

*Proof (ii).* Let  $\epsilon = \frac{p}{q}$ , such that  $p, q \in \mathbb{N}$ ,  $\gcd(p, q) = 1$ . Consider any  $W_1, W_2 \in \mathcal{W}_x$  such that  $c(W_1) < c(W_2) < c(W_1) + \frac{1}{q}$ . There exist  $x, y \in \mathbb{Z}$  such that  $c(W_2) - c(W_1) = x + y\epsilon$ . This implies that  $0 < x + y\frac{p}{q} < \frac{1}{q}$ . So,  $0 < xq + yp < 1$ . But  $xq + yp$  is an integer. Contradiction.

Due to lemma 3(i) the solution with the largest cost and the solution with the least cost in  $\mathcal{W}_x$  differ by no more than  $2 + \epsilon$ . Hence, there can be only  $(2 + \epsilon)q = O(1)$  distinct equivalence classes of equal cost in  $\mathcal{W}_x$ .  $\square$

*Proof (iii).* Consider any  $W \in \mathcal{W}_x$ . We know, from lemma 2(ii), that in  $W$  there may be no more than 3 walks that cross point  $x$ . Let us consider 3 chains  $C_1, C_2$  and  $C_3$  that correspond to the walks that cross  $x$ . In addition let  $|C_1| \leq |C_2| \leq |C_3|$ . If there aren't sufficient chains, let the smallest chains be empty chains, with  $p(C_1) = x$  and  $|C_1| = 0$ , for example.

We claim that  $|C_1| \leq 4$ . As a consequence of lemma 1(ii) there must be a full read in the interval  $[x - 4, x)$ . Lemma 2(iv) states that all three chains cannot go beyond this read. Hence,  $|C_1| \leq 4$ . Lemma 1(i) states that there are no more than  $8 \cdot 4 = 32$  reads in  $[x - 4, x)$ . Lemma 2(iii) implies that  $C_1$  can possibly start at at most 32 different positions in  $[x - 4, x)$ . In other words there are no more than 32 different positions in  $[x, x + 1)$  where  $C_1$  can end. Let us call this set of positions  $P_1$ .  $|P_1| \leq 32$ .

Now look at the set of possible endpoints of the chain  $C_3$ . Lemmas 2(iii) and 1(i) imply that  $C_3$  may start at no more than  $8L$  different positions and therefore end at no more than  $8L$  different positions. Let us call this set of positions  $P_3$ .  $|P_3| \leq 8L$ .

There are 3 possible directions that the last walk in any chain can take and a 4th possibility that that chain does not exist. Let us call the set of all different possible directions of the last reads of any  $W$  by the set  $Q$ .  $|Q| \leq 4^3 = 64$ .

From lemma 3(ii) we know that there is a constant sized set of possible costs for any solution in  $\mathcal{W}_x$ . Let us call this set of costs  $K$ .  $|K| = O(1)$ .

We can therefore classify any  $W \in \mathcal{W}_x$  by a set of equivalence classes. The set of classes being  $T = P_1 \times P_3 \times Q \times K$ .  $|T| = O(L)$ . Next we show that there are 0 or 1 nonredundant solutions in each class. We define the *characteristic*,  $t_x(\cdot)$

of a feasible partial solution at  $x$  as the equivalence class to which it belongs, i.e.  $t_x(W) \in T$ .

Consider two partial feasible solutions  $W_1, W_2 \in \mathcal{W}_x$ , such that  $t_x(W_1) = t_x(W_2)$ . The only thing that the two solutions differ in, is the end position of chain  $C_2$ . Let  $u$  be the end point of chain  $C_2$  in  $W_1$  and let  $v$  be the end point of  $C_2$  in  $W_2$ . Wlog, let  $u > v$ . Say, that  $W_2$ , goes on to form an optimum solution. Simply take every walk that was added to  $W_2$  after  $x$  and place it in  $W_1$ . Clearly,  $W_1$  must also form an optimum. So,  $W_2$  is expendable. We say that  $W_1$  *dominates*  $W_2$ . Hence  $|\mathcal{W}_x| = O(L)$ .  $\square$

### 3.4 The Algorithm

We introduce some additional notation before presenting the algorithm. As defined in lemma 3(iii), any partial feasible solution  $W \in \mathcal{W}_x$  has a *characteristic*,  $t_x(W)$ . The characteristic is a tuple which specifies the directions of the reads that cross  $x$ , the cost  $c(W)$ , end position of the longest chain that crosses  $x$  and the end position of the shortest chain that crosses  $x$ . If there are only one or two chains that cross  $x$  then the end position of the shortest chain is taken to be  $x$ . If there are no chains that cross  $x$  then both shortest and longest end positions are set to  $x$ . If  $W_1, W_2 \in \mathcal{W}_x$ ,  $W_1 \neq W_2$ ,  $t_x(W_1) = t_x(W_2)$  then by the construction in lemma 3(iii), one of the solutions  $W_1$  or  $W_2$  is expendable. So, if we say that  $W_1$  *dominates*  $W_2$  then  $W_2$  can be discarded.

If  $W \in \mathcal{W}_x$ , then there is some point in  $[x, x+1)$  at which coverage is inadequate. We define *augment*( $W$ ) to be the set of all possible ways of adding walks to  $W$  such that the interval  $[x, x+1)$  has adequate coverage. Note that all walks added are always flush with some walk or a read. Clearly, there can be no more than  $3^3 = 27$  different ways of doing this. We now present algorithm *Finish* in figure 3.

**Theorem 1.** *The algorithm  $\text{Finish}(S)$  finds a set of walks with least cost in  $O(nL)$  time.*

*Proof.* We first show that *Finish* finds a solution with optimum cost, by an induction on  $x$ . We want to show that if  $\mathcal{B}_x$  contains a solution which is a subset of an optimum then  $\mathcal{B}_{x+1}$  contains a solution which is a subset of an optimum. Say that  $\mathcal{B}_x$  contains a solution  $W$  that is a subset of some optimum. We know from lemma 3(i) that there can be no other  $W' \in \mathcal{B}_x$  such that  $c(W) > c(W') + 2 + \epsilon$ . If  $W$  is dominated by some other solution then even though  $W \notin \mathcal{W}_x$ , there is some solution  $W'' \in \mathcal{W}_x$  which dominates  $W$  and hence it must be the case that  $W''$  can form an optimum solution. As long as there is a feasible solution that is a subset of some optimum in  $\mathcal{W}_x$  one of its augmented solutions must also be a subset of some optimum solution in  $\mathcal{B}_{x+1}$ . In the base case  $\mathcal{W}_0 = \{\phi\}$ . An empty solution is a subset of any optimum solution. Hence, *Finish* finds a solution with optimum cost.

Next we show that the running time is  $O(nL)$ . The outermost loop runs for  $O(n)$  rounds. In any round,  $x$ , the first **for** loop runs in  $|\mathcal{B}_x|$  time. The second **for**

*Algorithm Finish*(S)

```

1:  $\mathcal{B}_0 \leftarrow \{\phi\}$ 
2: for  $x \leftarrow 0$  to  $n - 1$ 
3:    $\mathcal{B}_{x+1} \leftarrow \{\phi\}$ 
4:    $c^* \leftarrow \min_{W \in \mathcal{B}_x} \{c(W)\}$ 
5:   for every  $W \in \mathcal{B}_x$ 
6:     if  $c(W) > 2 + \epsilon + c^*$  then  $\mathcal{B}_x \leftarrow \mathcal{B}_x \setminus \{W\}$ 
7:   end for
8:   for every  $W_1, W_2 \in \mathcal{B}_x$  such that  $t_x(W_1) = t_x(W_2)$ 
9:     if  $W_1$  dominates  $W_2$  then  $\mathcal{B}_x \leftarrow \mathcal{B}_x \setminus \{W_2\}$ 
10:  end for
11:   $\mathcal{W}_x \leftarrow \mathcal{B}_x$ 
12:  for every  $W \in \mathcal{W}_x$ 
13:     $\mathcal{B}_{x+1} \leftarrow \mathcal{B}_{x+1} \cup \text{augment}(W, x)$ 
14:  end for
15: end for
16: return  $\operatorname{argmin}_{W \in \mathcal{B}_n} \{c(W)\}$ 

```

**Fig. 3.** Finish Algorithm

loop, though it looks quadratic can be done in linear time. We can group every solution by its characteristic. Then for each characteristic keep the one with the longest middle chain. So, this loop takes no more than  $|\mathcal{B}_x| + c_1 L$  time for some constant  $c_1$  (related to the number of equivalence classes). In the third **for** loop augment adds no more than 27 new solutions for each solution in  $\mathcal{W}_x$ . Hence it takes  $27|\mathcal{W}_x|$  time. So, the total time taken is  $\sum_{x=0}^{n-1} (2|\mathcal{B}_x| + 27|\mathcal{W}_x| + c_1 L)$ . Note that the sets  $\mathcal{B}_x$  are constructed from elements of  $\mathcal{W}_{x-1}$ .  $|\mathcal{B}_x| \leq 27|\mathcal{W}_{x-1}|$ . Total time is  $O(nL) + \sum_{x=0}^{n-1} 81|\mathcal{W}_x|$ . From lemma 3(iii),  $|\mathcal{W}_x| = O(L)$ . Hence *Finish* runs in worst case time  $O(nL)$   $\square$

Since the length  $L$  cannot take a value larger than  $n$  *Finish* runs in worst case  $O(n^2)$  time. It can be shown that if the problem instance is random and the number of reads is  $O(n)$  then the length of the longest chain is constant with high probability. This implies a linear time algorithm. This randomized model is discussed in [Har02] and extends naturally to our algorithm without any modification.

## 4 Divide and Conquer

*Finish* can be speeded up by localizing the computations. We show that making the computation too fine grained or too coarse slows down the algorithm. This is an adaptation of the technique used by Hart [Har02]. We split the interval in the problem instance into  $m$  equally sized subintervals. We borrow terminology from [Har02] and call each of these *divisions*. There are  $\lceil \frac{n}{m} \rceil$  divisions. So, the first division is the interval  $[0, m)$ , the second division is  $[m, 2m)$  and so on. It follows

*Algorithm Finish2*

```

1:  $\mathcal{W}_0 \leftarrow \{\phi\}$ 
2: for  $j \leftarrow 0$  to  $\lfloor \frac{n}{m} \rfloor$ 
3:   for each divisional characteristic in  $\mathcal{W}_{mj}$  select one  $W \in \mathcal{W}_{mj}$ 
4:      $P \leftarrow$  the set of walks in  $W$  that cross  $mj$ .
5:     call Finish( $D_j \cup P$ )
6:     let  $\mathcal{W}$  be the set of feasible solutions returned by Finish
7:     reconstruct all possible feasible solutions from  $\mathcal{W}$  (lemma 4), let these be
        $\mathcal{B}_{m(j+1)}$ .
8:     create  $\mathcal{W}_{m(j+1)}$  from  $\mathcal{B}_{m(j+1)}$  according to lemma 3 by discarding unnecessary
       solutions.
9:   end for
10: end for

```

**Fig. 4.** Algorithm *Finish2*

from lemma 1 that there are  $O(m)$  reads in each division. We index divisions with the subscript  $j$ . For the  $j$ th division, let  $D_j$  be the set of all reads that have a point in common with  $[mj, mj + m)$ . Let  $f_{ij}$  be the fractional part of the real number  $p(r_i)$  for each  $r_i \in D_j$ . The subscript  $i$  orders reads  $r_i$  in ascending order of  $p(r_i)$ . This means that  $0 \leq f_{0,j} \leq f_{1,j} \leq f_{2,j} \cdots \leq 1$ . These fractional values define small intervals. Each of these small intervals is called a *fraction*. There are  $O(m)$  fractions corresponding to any division. We can now partition the set of all feasible partial solutions  $\mathcal{W}_{mj}$  into equivalence classes.

Recall that  $\mathcal{W}_x$  is the set of feasible partial solutions such that for each solution in the set every point before  $x$ , on the real line is adequately covered. Since we are at the beginning of the  $j$ th division we set  $x = mj$ .  $\mathcal{W}_x$  is partitioned into equivalence classes by the characteristic function  $t_x(\cdot)$ . Along the same lines we define a new characteristic function  $d_x(\cdot)$  that partitions  $\mathcal{W}_x$  into equivalence classes with respect to the  $j$ th division, we call this the *divisional characteristic*. Intuitively two feasible partial solutions belonging to the same divisional characteristic can be treated as one solution for the entire division. Consider any solution,  $W \in \mathcal{W}_x$ . At most 3 walks protrude beyond  $x$  in this solution. The fractional part of the positions of each of these 3 walks lies in one of the *fractions* of this division. Each solution can be categorized by which set of 3 fractions its last three walks fall into. The solutions may be further categorized by the directions of the last three walks. These 6 parameters define the divisional characteristic for a feasible solution. From the proof of theorem 1 we know that the shortest of these three chains can end in at most  $O(1)$  positions. Therefore, each partial solution can have one of  $O(m^2)$  possible divisional characteristics.

**Lemma 4.** *If  $W_1$  and  $W_2$  are two feasible partial solutions in  $\mathcal{W}_{mj}$ , with identical divisional characteristics,  $d_{mj}(W_1) = d_{mj}(W_2)$ , and at the end of the division,  $W_1$  and  $W_2$  give rise to two sets of feasible partial solutions,  $\mathcal{S}_1$  and  $\mathcal{S}_2$  respectively, then  $\mathcal{S}_2$  can be constructed from  $\mathcal{S}_1$  in linear time (linear in the size of  $\mathcal{S}_1$ ).*



*Proof (Sketch).* Let algorithm *Finish* run on the feasible partial solution,  $W_1$  over the division  $[mj, mj + m)$ . At the end of this we have a set of feasible solutions that cover this division adequately. Let this set be  $\mathcal{S}_1$ . Similarly we get a set of solutions  $\mathcal{S}_2$  after running *Finish* on  $W_2$ . Let  $s_1$  be a solution in  $\mathcal{S}_1$ . We describe how to construct a feasible partial solution  $s_2 \in \mathcal{S}_2$  from  $s_1$ . All chains that start at reads in  $s_1$  are left as is. Any chain that starts at a walk that enters the division, is shifted so that it starts at its corresponding walk in  $W_2$  (there is a correspondence because  $W_1$  and  $W_2$  have the same divisional characteristic). Two chains that start anywhere in the same fraction remain in the same fraction. There is no change in read coverage within a fraction (only at the border of a fraction). Since the end point of each walk has the same coverage in both solutions the algorithm must make the same decision in both cases (albeit shifted by a small amount). Therefore,  $s_2$  is a feasible partial solution in  $\mathcal{S}_2$  which is obtained from  $s_1$  by a slight shift of the first few chains.

The procedure described takes a solution in  $\mathcal{S}_1$  and creates one that belongs to  $\mathcal{S}_2$ . Note that if we ran the procedure on  $s_2$  we would get back  $s_1$ . Therefore, this procedure is a bijective mapping from  $\mathcal{S}_1$  to  $\mathcal{S}_2$ . This means that we can reconstruct every solution in  $\mathcal{S}_2$  from a solution in  $\mathcal{S}_1$  and vice versa. The time needed is the size of the solution in the worst case.  $\square$

**Theorem 2.** *Algorithm Finish2 finds an optimum solution in  $O(n^{1\frac{3}{4}})$  time.*

*Proof.* The correctness of *Finish2* can be established with an induction argument analogous to the one used for *Finish* in conjunction with lemma 4.

There are a total of  $\frac{n}{m}$  divisions. On entering each division, there are a set of  $O(n)$  feasible partial solutions. This set of feasibles can be partitioned into  $O(m^2)$  equivalence classes. We run *Finish* for one member of each equivalence class. This takes a total of  $O(m^4)$  time. At the end of a division we have a total of not more than  $O(m^3)$  feasible solutions ( $O(m)$  solutions from each of  $O(m^2)$  runs). It takes linear time to reconstruct a global feasible solution from each of these, and since there are a total of  $O(n)$  global feasibles at the end of the division the total time needed is  $O(n + m^4)$ . Summing the time needed for entering a division, processing it and leaving it gives us  $O(n + m^4)$  per division. If we set  $m = n^{\frac{1}{4}}$  we get the fastest time to process a division. There are  $O(\frac{n}{m})$  divisions in all. Hence the total running time is  $O(n^{1\frac{3}{4}})$ .  $\square$

[ As a side note we mention that the sub-cubic algorithm in [Har02] appears to have a running time of  $O(n^{2\frac{3}{5}})$  instead of the claimed  $O(n^{2\frac{1}{3}})$  bound. The analysis is similar to the one used in theorem 2. In their case, there is an  $O(n^3)$  finish algorithm. The number of chains that any feasible partial solution ends with is at most  $O(n^2)$ . With these two facts in mind we redo the above analysis. There are  $\frac{n}{m}$  divisions. On entering each division there are  $O(n^2)$  feasible partial solutions. This set is partitioned into  $O(m^2)$  equivalence classes. The finish algorithm takes  $O(m^3)$  time on each of these cases. Hence, the total time spent on entry is  $O(m^5)$ . At the end of the division, there are  $O(m^4)$  feasible solutions ( $O(m^2)$  solutions from each of  $O(m^2)$  runs). Updating each solution takes  $O(m)$

time. There may be only  $O(n^2)$  solutions at the end of the division. So, the net time taken to finish a division is  $O(m^5 + n^2)$ . Setting  $m = n^{\frac{2}{5}}$  minimizes this time. There are  $\frac{n}{m}$  divisions, so the total running time should be  $O(n^{2\frac{3}{5}})$ . The algorithm is still sub-cubic. ]

## 5 Conclusion

We have described a new algorithm for finding the optimum set of walks to finish a DNA sequence with a worst case running time of  $O(n^{1\frac{3}{4}})$ . It is the first sub-quadratic algorithm for this problem. The algorithm can be made online in the sense that the reads appear in a left to right order and knowledge of  $n$  is not necessary.

We show that the number of feasible solutions at any point on the problem interval is no more than  $O(n)$ . This is an improvement over the known  $O(n^2)$ . We believe that this could be much smaller. Another direction for future work is in trying to reduce the  $O(n)$  update time required between two consecutive intervals. This may be possible with the help of a data structure suited to this problem. The running time of the algorithm has a strong dependence on  $\epsilon$ . It is very likely that there is an  $O(n^2 \text{ polylog } n)$  algorithm that is independent of  $\epsilon$ .

## References

- [CKM<sup>+</sup>00] Éva Czabarka, Goran Konjevod, Madhav V. Marathe, Allon G. Percus, and David C. Torney. Algorithms for optimizing production DNA sequencing. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-00)*, pages 399–408, New York, January 9–11 2000. ACM Press.
- [GAG98] D. Gordon, C. Abajian, and P. Green. Consed: A graphical tool for sequence finishing. *Genome Research*, 8:195–202, March 1998.
- [GDG01] D. Gordon, C. Desmarais, and P. Green. Automated finishing with autofinish. *Genome Research*, 4:614–625, April 2001.
- [Har02] David Hart. An optimal (expected time) algorithm for minimizing lab costs in DNA sequencing. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Algorithms (SODA-02)*, pages 885–893, New York, January 6–8 2002. ACM Press.
- [Hgp98] Human genome project, U.S. department of energy. *Human Genome News*, 9(3), July 1998.
- [PT99] Allon G. Percus and David C. Torney. Greedy algorithms for optimized DNA sequencing. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-99)*, pages 955–956, N.Y., January 17–19 1999. ACM-SIAM.

# Algorithms for Generating Minimal Blockers of Perfect Matchings in Bipartite Graphs and Related Problems<sup>\*</sup>

Endre Boros, Khaled Elbassioni, and Vladimir Gurvich

RUTCOR, Rutgers University, 640 Bartholomew Road, Piscataway NJ 08854-8003;  
`{boros,elbassio,gurvich}@rutcor.rutgers.edu`

**Abstract.** A minimal blocker in a bipartite graph  $G$  is a minimal set of edges the removal of which leaves no perfect matching in  $G$ . We give a polynomial delay algorithm for finding all minimal blockers of a given bipartite graph. Equivalently, this gives a polynomial delay algorithm for listing the anti-vertices of the perfect matching polytope  $P(G) = \{x \in \mathbb{R}^E \mid Hx = \mathbf{e}, x \geq 0\}$ , where  $H$  is the incidence matrix of  $G$ . We also give similar generation algorithms for other related problems, including  $d$ -factors in bipartite graphs, and perfect 2-matchings in general graphs.

## 1 Introduction

Let  $G = (A, B, E)$  be a bipartite graph with bipartition  $A \cup B$  and edge set  $E$ . For subsets  $A' \subseteq A$  and  $B' \subseteq B$ , denote by  $(A', B')$  the subgraph of  $G$  induced by these sets. A perfect matching in  $G$  is a subgraph in which every vertex in  $A \cup B$  has degree exactly one, or equivalently a subset of  $|A| = |B|$  pairwise disjoint edges of  $G$ . Throughout the paper, we shall always assume that a graph with no vertices has a perfect matching. A blocker in  $G$  is a subset of edges  $X \subseteq E$  such that the graph  $G(A, B, E \setminus X)$  does not have any perfect matching. A blocker  $X$  is minimal if, for every edge  $e \in X$ , the set  $X \setminus \{e\}$  is not a blocker. Denote respectively by  $\mathcal{M}(G)$  and  $\mathcal{B}(G)$  the families of perfect matchings and minimal blockers for  $G$ . Note that in any bipartite graph  $G$ , the set  $\mathcal{B}(G)$  is the family of minimal transversals to the family  $\mathcal{M}(G)$ , i.e.  $\mathcal{B}(G)$  is the family of minimal edge sets containing an edge from every perfect matching in  $G$ . Clearly, the above definitions imply that if  $|A| \neq |B|$ , then  $\mathcal{B}(G) = \{\emptyset\}$ .

The main problem we consider in this paper is the following.

**GEN( $\mathcal{B}(G)$ ):** *Given a bipartite graph  $G$ , enumerate all minimal blockers of  $G$ .*

The analogous problem GEN( $\mathcal{M}(G)$ ) of enumerating perfect matchings for bipartite graphs was considered in [9,20].

---

<sup>\*</sup> This research was supported by the National Science Foundation (Grant IIS-0118635). The third author is also grateful for the partial support by DIMACS, the National Science Foundation's Center for Discrete Mathematics and Theoretical Computer Science.

In such generation problems the output size may be exponential in the input size, therefore the efficiency of an algorithm is evaluated by measuring the time it needs to find a *next* element, see e.g., [14]. More precisely, we say that the elements of a set  $\mathcal{S}$  represented in some implicit way by the input  $I$  are generated with *polynomial delay*, if the generation algorithm produces all elements of  $\mathcal{S}$  in some order such that the computational time between any two outputs is limited by a polynomial expression of the input size. The generation is said to be *incrementally polynomial*, if the time needed to find the  $k + 1$ st element, after the generation of the first  $k$  elements, depends polynomially not only on the input size but also on  $k$ .

The problems of generating perfect matchings and minimal blockers are special cases of the more general open problems of generating vertices and anti-vertices of a polytope given by its linear description. Indeed, let  $H$  be the vertex-edge incidence matrix of  $G = (A, B, E)$ , and consider the polytope  $P(G) = \{x \in \mathbb{R}^E \mid Hx = \mathbf{e}, x \geq 0\}$ , where  $\mathbf{e} \in \mathbb{R}^{|A|+|B|}$  is the vector of all ones. Then the perfect matchings of  $G$  are in one-to-one correspondence with the vertices of  $P(G)$ , which in turn, are in one-to-one correspondence with the minimal collections of columns of  $H$  containing the vector  $\mathbf{e}$  in their conic hull. On the other hand, the minimal blockers of  $G$  are in one-to-one correspondence with the anti-vertices of  $P(G)$ , i.e. the maximal collections of columns of  $H$ , not containing  $\mathbf{e}$  in their conic hull. Both corresponding generating problems are open in general, see [2] for details. In the special case, when  $H$  is the incidence matrix of a bipartite graph  $G$ , the problem of generating the vertices of  $P(G)$  can be solved with polynomial delay [9,20]. In this note, we obtain an analogous result for anti-vertices of  $P(G)$ .

**Theorem 1.** *Problem  $\text{GEN}(\mathcal{B}(G))$  can be solved with polynomial delay.*

For non-bipartite graphs  $G$ , it is well-known that the vertices of  $P(G)$  are half-integral [16] (i.e. the components of each vertex are in  $\{0, 1, 1/2\}$ ), and that they correspond to the basic 2-matchings of  $G$ , i.e. subsets of edges that cover the vertices with vertex-disjoint edges and vertex-disjoint odd cycles. Polynomial delay algorithms exist for listing the vertices of 0/1-polytopes, simple and simplicial polytopes [1,5], but the status of the problem for general polytopes is still open. We show here that for the special case of the polytope  $P(G)$  of a graph  $G$ , the problem can be solved in incremental polynomial time.

**Theorem 2.** *All basic perfect 2-matchings of a graph  $G$  can be generated in incremental polynomial time.*

The proof of Theorem 1 is based on a nice characterization of minimal blockers, which may be of independent interest. The method used for enumeration is the supergraph method which will be outlined in Section 3.1. A special case of this method, when applicable, can provide enumeration algorithms with stronger properties, such as enumeration in lexicographic ordering, or enumeration with polynomial space. This special case will be described in Section 3.2 and will be used in Section 5 to solve two other related problems of enumerating  $d$ -factors

in bipartite graphs, and enumerating perfect 2-matchings in general graphs. The latter result will be used to prove Theorem 2.

## 2 Properties of Minimal Blockers

### 2.1 The Neighborhood Function

Let  $G = (A, B, E)$  be a bipartite graph. For a subset  $X \subseteq A \cup B$ , denote by  $\Gamma(X) = \Gamma_G(X) = \{v \in (A \cup B) \setminus X \mid \{v, x\} \in E \text{ for some } x \in X\}$  the set of neighbors of  $X$ . It is known (see, e.g., [15]), and also easy to verify, that set-function  $|\Gamma(\cdot)|$  is submodular, i.e.  $|\Gamma(X \cup Y)| + |\Gamma(X \cap Y)| \leq |\Gamma(X)| + |\Gamma(Y)|$  holds for all subsets  $X, Y \subseteq A \cup B$ . A simple but useful observation about submodular set-functions is the following (see, e.g., [17]):

**Proposition 1.** *Let  $V$  be a finite set and  $f : 2^V \mapsto \mathbb{R}$  be a submodular set-function. Then the family of sets  $X$  minimizing  $f(X)$  forms a sub-lattice of the Boolean lattice  $2^V$ . If  $f(X)$  is polynomial-time computable for every  $X \subseteq V$ , then the unique minimum and maximum of this sub-lattice can be computed in polynomial time.*

### 2.2 Matchable Graphs

Matchable graphs, a special class of bipartite graphs with a *positive surplus* (see Section 1.3 in [16] for definitions and properties), were introduced in [12]. They play an important role in the characterization of minimal blockers.

**Definition 1 ([12]).** *A bipartite graph  $G = (A, B, E)$  with  $|A| = |B| + 1$  is said to be matchable if the graph  $G \setminus v$  has a perfect matching for every vertex  $v \in A$ .*

The smallest matchable graph is one in which  $|A| = 1$  and  $|B| = 0$ . The following is a useful equivalent characterization of matchable graphs.

**Proposition 2 ([12]).** *For a bipartite graph  $G = (A, B, E)$  with  $|A| = |B| + 1$ , the following statements are equivalent:*

- (M1)  *$G$  is matchable;*
- (M2)  *$G$  has a matchable spanning tree, that is, a spanning tree  $T$  in which every node  $v \in B$  has degree 2.*

It can be shown that for bipartite graphs  $G = (A, B, E)$  with  $|A| = |B| + 1$  these properties are further equivalent with

- (M3)  *$G$  contains a unique independent set of size  $|A|$ , namely the set  $A$ .*

Given a bipartite graph  $G$ , let us say that a matchable subgraph of  $G$  is *maximal*, if it is not properly contained in any other matchable subgraph of  $G$ . The following property can be derived from a result by Dulmage and Mendelsohn, or more generally from the Gallai-Edmonds Structure Theorem (see Section 3.2 in [16] for details and references).

**Proposition 3.** *Let  $G = (A, B, E)$  be a matchable graph with  $|A| = |B| + 1$ , and let  $a \in A$  and  $b \in B$  be given vertices. Then the graph  $G' = G - \{a, b\}$  is the union of a maximal matchable subgraph  $G_1 = (A_1, B_1)$  of  $G'$  and a (possibly empty) subgraph  $G_2 = (A_2, B_2)$  which has a perfect matching, such that the graph  $(A_1, B_2)$  is empty. Furthermore, this decomposition of  $G'$  into  $G_1$  and  $G_2$  is unique, and can be found in polynomial time.*

The next proposition acts, in a sense, as the opposite to Proposition 3, and can be derived in a similar way: Given a decomposition of a matchable graph into a matchable subgraph and a subgraph with a perfect matching, we can reconstruct in a *unique* way the original matchable graph.

**Proposition 4.** *Let  $G = (A, B, E)$  be a bipartite graph and  $G_1 = (A_1, B_1)$  and  $G_2 = (A_2, B_2)$  be two vertex-disjoint subgraphs of  $G$  such that  $G_1$  is matchable,  $G_2$  has a perfect matching, and  $A = A_1 \cup A_2$  and  $B = B_1 \cup B_2$ . Then there is a unique extension of  $G_1$  into a maximal matchable subgraph  $G'_1 = (A'_1, B'_1)$  of  $G$  such that the graph  $G'_2 \stackrel{\text{def}}{=} (A_1 \cup A_2 \setminus A'_1, B_1 \cup B_2 \setminus B'_1)$  has a perfect matching. Such an extension is the unique possible decomposition of  $G$  into a maximal matchable graph plus a graph with a perfect matching, and can be computed in polynomial time.*

As a simple application of Proposition 4, we obtain the following corollary.

**Corollary 1.** *Let  $G = (A, B, E)$  be a bipartite graph and  $G' = (A', B')$  be a matchable subgraph of  $G$ . Given two vertices  $a \in A \setminus A'$  and  $b \in B \setminus B'$  such that there is an edge  $\{b, a'\}$  for some  $a' \in A'$ , there exists a unique decomposition of  $G' = (A' \cup \{a\}, B' \cup \{b\})$  into a maximal matchable graph plus a graph with a perfect matching.*

The next property, which can be derived analogously to Proposition 3, implies that matchable graphs are “continuous” in the sense that we can reach from a given matchable graph to any matchable graph containing it by repeatedly appending pairs of vertices, one at a time, always remaining within the class of matchable graphs.

**Proposition 5.** *Let  $T = (A, B, E)$  and  $T' = (A', B', E')$  be two matchable spanning trees such that  $|A| = |B| + 1$ ,  $|A'| = |B'| + 1$ ,  $A \subseteq A'$  and  $B \subseteq B'$ . Then (i) there exists a vertex  $b \in B' \setminus B$  such that  $|\Gamma_{T'}(\{b\}) \cap A| \geq 1$ , (ii) for any such vertex  $b$ , there exists a vertex  $a \in A' \setminus A$ , such that the graph  $G[a, b] \stackrel{\text{def}}{=} (A \cup \{a\}, B \cup \{b\}, E \cup E')$  is matchable.*

The above properties readily imply the following corollary.

**Corollary 2.** *Let  $G = (A, B, E)$  and  $G' = (A', B', E')$  be two matchable graphs such that  $|A| = |B| + 1$ ,  $|A'| = |B'| + 1$ ,  $A \subset A'$ , and  $B \subset B'$ . Then there exists a pair of vertices  $a \in A' \setminus A$  and  $b \in B' \setminus B$  such that the graph  $(A \cup \{a\}, B \cup \{b\}, E \cup E')$  is matchable.*

To arrive to a useful characterization of minimal blockers, we need one more definition.

**Definition 2.** Let  $G = (A, B, E)$  be a bipartite graph having a perfect matching. A *matchable split* of  $G$ , denoted by  $[(A_1, B_1), (A_2, B_2), (A_3, B_3)]$ , is a partition of the vertex set of  $G$  into sets  $A_1, A_2, A_3 \subseteq A$  and  $B_1, B_2, B_3 \subseteq B$  such that

- (B1)  $|A_1| = |B_1| + 1$ ,  $|A_2| = |B_2| - 1$ ,  $|A_3| = |B_3|$ ,
- (B2) the graph  $G_1 = (A_1, B_1)$  is matchable,
- (B3) the graph  $G_2 = (A_2, B_2)$  is matchable,
- (B4) the graph  $G_3 = (A_3, B_3)$  has a perfect matching, and
- (B5) the graphs  $(A_1, B_3)$  and  $(A_3, B_2)$  are empty.

Denote by  $\mathcal{S}(G)$  the family of all matchable splits of  $G$ . We are now ready to state our characterization for minimal blockers in bipartite graphs.

**Lemma 1.** Let  $G = (A, B, E)$  be a bipartite graph in which there exists a perfect matching. Then a subset of edges  $X \subseteq E$  is a minimal blocker if and only if there is a matchable split  $[(A_1, B_1), (A_2, B_2), (A_3, B_3)]$  of  $G$ , such that  $X = \{\{a, b\} \in E : a \in A_1, b \in B_2\}$  is the set of all edges between  $A_1$  and  $B_2$  in  $G$ . This correspondence between minimal blockers and matchable splits is one-to-one. Given one representation we can compute the other in polynomial time.

### 3 Enumeration Algorithms

Given a (bipartite) graph  $G = (V, E)$ , consider the problem of listing all subgraphs of  $G$ , or correspondingly, the family  $\mathcal{F}_\pi \subseteq 2^E$  of all minimal subsets of  $E$ , satisfying some monotone property  $\pi : 2^E \mapsto \{0, 1\}$ . For instance, if  $\pi(X)$  is the property that the subgraph with edge set  $X \subseteq E$  has a perfect matching, then  $\mathcal{F}_\pi$  is the family of perfect matchings of  $G$ . Enumeration algorithms for listing subgraphs satisfying a number of monotone properties are well known. For instance, it is known [19] that the problems of listing all minimal cuts or all spanning trees of an undirected graph  $G = (V, E)$  can be solved with delay  $O(|E|)$  per generated cut or spanning tree. It is also known (see e.g., [7,10,18]) that all minimal  $(s, t)$ -cuts or  $(s, t)$ -paths, can be listed with delay  $O(|E|)$  per cut or path. Furthermore, polynomial delay algorithms also exist for listing perfect matchings, maximal matchings, maximum matchings in bipartite graphs, and maximal matchings in general graphs, see e.g. [9,20,21].

In the next subsections we give an overview of two commonly used techniques for solving such enumeration problems.

#### 3.1 The Supergraph Approach

This technique works by building and traversing a directed graph  $\mathcal{G} = (\mathcal{F}_\pi, \mathcal{E})$ , defined somehow on the set of elements of the family to be generated  $\mathcal{F}_\pi \subseteq 2^E$ . The arcs of  $\mathcal{G}$  are defined by a *polynomial-time computable* neighborhood function  $\mathcal{N} : \mathcal{F}_\pi \mapsto 2^{\mathcal{F}_\pi}$  that defines, for any  $X \in \mathcal{F}_\pi$  the set of its outgoing neighbors  $\mathcal{N}(X)$  in  $\mathcal{G}$ . A special vertex  $X_0 \in \mathcal{F}_\pi$  is identified from which all other vertices of  $\mathcal{G}$  are reachable. The method works by performing a (breadth- or depth-first search) traversal on the nodes of  $\mathcal{G}$ , starting from  $X_0$ . The following is a basic fact about this approach:



**(S1)** If  $\mathcal{G}$  is strongly connected and  $|\mathcal{N}(X)| \leq p(|\pi|)$  for every  $X \in \mathcal{F}_\pi$ , where  $p(|\pi|)$  is a polynomial that depends only on the size of the description of  $\pi$ , then the supergraph approach gives us a polynomial delay algorithm for enumerating  $\mathcal{F}_\pi$ , starting from an arbitrary set  $X_0 \in \mathcal{F}_\pi$ .

Let us consider as an example the *generation of minima of submodular functions*: Let  $f : 2^V \mapsto \mathbb{R}$  be a submodular function,  $\alpha = \min\{f(X) : X \subseteq V\}$ , and  $\pi(X)$  be the property that  $X \subseteq V$  contains a minimizer of  $f$ . By Proposition 1, the set  $\mathcal{F}_\pi$  forms a sub-lattice  $\mathcal{L}$  of  $2^V$ , and the smallest element  $X_0$  of this lattice can be computed in polynomial time. For  $X \in \mathcal{F}_\pi$ , define  $\mathcal{N}(X)$  to be the set of subsets  $Y \in \mathcal{L}$  that immediately succeed  $X$  in the lattice order. The elements of  $\mathcal{N}(X)$  can be computed by finding, for each  $v \in V \setminus X$ , the smallest cardinality minimizer  $X' = \operatorname{argmin}\{f(Y) : Y \supseteq X \cup \{v\}\}$  and checking if  $f(X') = \alpha$ . Then  $|\mathcal{N}(X)| \leq |V|$ , for all  $X \in \mathcal{F}_\pi$ . Note also that the definition of  $\mathcal{N}(\cdot)$  implies that the supergraph is strongly connected. Thus (S1) implies that all the elements of  $\mathcal{F}_\pi$  can be enumerated with polynomial delay.

### 3.2 The Flashlight Approach

This can be regarded as an important special case of the supergraph method. Assume that we have fixed some order on the elements of  $E$ . Let  $X_0$  be the lexicographically smallest element in  $\mathcal{F}_\pi$ . For any  $X \in \mathcal{F}_\pi$ , let  $N(X)$  consist of a single element, namely, the element next to  $X$  in lexicographic ordering. Thus the supergraph  $\mathcal{G}$  in this case is a Hamiltonian path on the elements of  $\mathcal{F}_\pi$  (see [19] for general background on backtracking algorithms). The following is a sufficient condition for the operator  $\mathcal{N}(\cdot)$  (and also for the element  $X_0$ ) to be computable in polynomial time:

**(F1)** For any two disjoint subsets  $S_1, S_2$  of  $E$ , we can check in polynomial time  $p(|\pi|)$  if there is an element  $X \in \mathcal{F}_\pi$ , such that  $X \supseteq S_1$  and  $X \cap S_2 = \emptyset$ .

The traversal of  $\mathcal{G}$ , in this case, can be organized in a backtracking tree of depth  $|E|$ , whose leaves are the elements of  $\mathcal{F}_\pi$ , as follows. Each node of the tree is identified with two disjoint subsets  $S_1, S_2 \subseteq E$ , and have at most two children. At the root of the tree, we have  $S_1 = S_2 = \emptyset$ . The left child of any node  $(S_1, S_2)$  of the tree at level  $i$  is  $(S_1 \cup \{i\}, S_2)$  provided that there is an  $X \in \mathcal{F}_\pi$ , such that  $X \supseteq S_1 \cup \{i\}$  and  $X \cap S_2 = \emptyset$ . The right child of any node  $(S_1, S_2)$  of the tree at level  $i$  is  $(S_1, S_2 \cup \{i\})$  provided that there is an  $X \in \mathcal{F}_\pi$ , such that  $X \supseteq S_1$  and  $X \cap (S_2 \cup \{i\}) = \emptyset$ . Furthermore, we may restrict our attention to subsets  $S_1$  satisfying certain properties. More precisely, let  $\mathcal{F}'_\pi \subseteq 2^E$  be a family of sets, containing  $\mathcal{F}_\pi$ , such that we can test in polynomial time if a set  $X$  belongs to it, and such that for every  $X \in \mathcal{F}_\pi$ , there is a set  $S \in \mathcal{F}'_\pi$  contained in  $X$ . Then the following is a weaker requirement than that of (F1):

**(F2)** For any two disjoint subsets  $S_1 \in \mathcal{F}'_\pi$  and  $S_2 \subseteq E$ , and given element  $i \in E \setminus (S_1 \cup S_2)$  such that  $S_1 \cup \{i\} \in \mathcal{F}'_\pi$ , we can check in polynomial time  $p(|\pi|)$  if there is an element  $X \in \mathcal{F}_\pi$ , such that  $X \supseteq S_1 \cup \{i\}$  and  $X \cap S_2 = \emptyset$ .



This way, under assumption (F2), we get a polynomial delay, polynomial space algorithm for enumerating the elements of  $\mathcal{F}_\pi$  in lexicographic order.

As an example, let us consider the *generation of maximal matchable subgraphs*: Let  $G = (A, B, E)$  be a bipartite graph. For subsets  $A' \subseteq A$  and  $B' \subseteq B$ , let  $\pi(A', B')$  be the property that the graph  $(A', B')$  contains a maximal matchable subgraph. To generate the family of maximal matchable graphs  $\mathcal{F}_\pi$ , we introduce an artificial element  $*$ , and consider the ground set  $E' = \{(a, b) : a \in A, b \in B \cup \{*\}\}$ . Any  $X \subseteq E'$  represents an induced subgraph  $G(X) = (A', B')$  of  $G$  where  $A' = \{a : (a, b) \in X\}$  and  $B' = \{b : (a, b) \in X, b \neq *\}$ . Furthermore, let us say that the elements of  $X \subseteq E'$  are disjoint if for every pair of elements  $(a, b), (a', b') \in X$ , we have  $a \neq a'$  and  $b \neq b'$ . Let  $\mathcal{F}'_\pi = \{X \subseteq E' : \text{the elements of } X \text{ are disjoint, one of them contains } *, \text{ and } G(X) \text{ is matchable}\}$ . Given two disjoint sets of pairs  $S_1, S_2 \subseteq E'$ , such that  $S_1 \in \mathcal{F}'_\pi$ , and a pair  $(a, b) \in E' \setminus (S_1 \cup S_2)$ , the check in (F2) can be performed in polynomial time. Indeed, by Corollary 2, all what we need to check is that the graph on  $S_1 \cup \{a, b\}$  is matchable.

## 4 Polynomial Delay Generation of Minimal Blockers

In this section, we use the supergraph approach to show that minimal blockers can be enumerated with polynomial delay. Using Lemma 1, we may equivalently consider the generation of matchable splits. We start by defining the neighborhood function used for constructing the supergraph  $\mathcal{G}_S$  of matchable splits.

### 4.1 The Supergraph

Let  $G = (A, B, E)$  be a bipartite graph that has a perfect matching. Given a matchable split  $X = [G_1 = (A_1, B_1), G_2 = (A_2, B_2), G_3 = (A_3, B_3)] \in \mathcal{S}(G)$ , the set of out-going neighbors of  $X$  in  $\mathcal{G}_S$  are defined as follows. For each edge  $\{a, b\} \in E$  connecting a vertex  $a \in A_2$  to a vertex  $b \in B_2$ , such that  $b$  is also connected to some vertex  $a' \in A_1$ , there is a unique neighbor  $X' = \mathcal{N}(X, a, b)$  of  $X$ , obtained by the following procedure:

1. Let  $G''_1 = (A_1 \cup \{a\}, B_1 \cup \{b\})$ . Because of the edges  $\{a, b\}, \{a', b\} \in E$ , the graph  $G''_1$  is matchable, see Corollary 1.
2. Delete the vertices  $a, b$  from  $G_2$ . This splits the graph  $G_2 - \{a, b\}$  in a unique way into a matchable graph  $G'_2 = (A'_2, B'_2)$  and a graph with a perfect matching  $G''_2 = (A''_2, B''_2)$  such that there are no edges in  $E$  between  $A''_2$  and  $B'_2$ , see Proposition 3.
3. Let  $G''_3 = G_3 \cup G''_2$ . Then the graph  $G''_3$  has a perfect matching. Using Proposition 4, extend  $G''_1$  in the union  $G''_1 \cup G''_3 = (A''_3, B''_3)$  into a maximal matchable graph  $G'_1 = (A'_1, B'_1)$  such that the graph  $G'_3 = (A'_3, B'_3) = (A''_3 \setminus A'_1, B''_3 \setminus B'_1)$  has a perfect matching.
4. Let  $X' = [(A'_1, B'_1), (A'_2, B'_2), (A'_3, B'_3)]$ , and note that  $X' \in \mathcal{S}(G)$ .

Similarly, for each edge  $\{a, b\} \in E$  connecting a vertex  $a \in A_1$  to a vertex  $b \in B_1$ , such that  $a$  is also connected to some vertex  $b' \in B_2$ , there is a unique neighbor  $X'$  of  $X$ , obtained by a procedure similar to the above.

## 4.2 Strong Connectivity

For the purpose of generating minimal blockers in a bipartite graph  $G = (A, B, E)$ , we may assume without loss of generality that

- (A1) The graph  $G$  is connected.
- (A2) Every edge in  $G$  appears in a perfect matching.

It is easy to see that both properties can be checked in polynomial time (see *elementary bipartite graphs* in [16]). Indeed, if  $G$  has a number of connected components, then the set of minimal blockers of  $G$  is the union of the sets of minimal blockers in the different components, computed individually for each component. Furthermore, all the edges in  $G$  that do not appear in any perfect matching can be removed (in polynomial time), since they do not appear in any minimal blocker. In this section, we prove the following.

**Lemma 2.** *Under the assumptions (A1) and (A2), the supergraph  $\mathcal{G}_S$  is strongly connected.*

Clearly Lemma 2 implies Theorem 1, according to (S1). Call the set of edges connected to a given vertex  $v \in A \cup B$  a  $v$ -star, and it by denote by  $v^*$ . We start with the following lemma.

**Lemma 3.** (i) *Under the assumption (A1) and (A2), every star is a minimal blocker.* (ii) *The matchable split corresponding to an  $a$ -star,  $a \in A$ , is  $[(\{a\}, \emptyset), (A \setminus \{a\}, B), (\emptyset, \emptyset)]$ .* (iii) *The matchable split corresponding to a  $b$ -star,  $b \in B$ , is  $[(A, B \setminus \{b\}), (\emptyset, \{b\}), (\emptyset, \emptyset)]$ .*

Given two matchable splits  $X = [(A_1, B_1), (A_2, B_2), (A_3, B_3)]$  and  $X' = [(A'_1, B'_1), (A'_2, B'_2), (A'_3, B'_3)]$ , let us say that  $X$  and  $X'$  are *perfectly nested* if

- (N1)  $A_1 \subseteq A'_1$  and  $B_1 \subseteq B'_1$ ,
- (N2)  $A_2 \supseteq A'_2$  and  $B_2 \supseteq B'_2$ , and
- (N3) each of the graphs  $(A_2 \cap A'_1, B_2 \cap B'_1)$ ,  $(A_3 \cap A'_1, B_3 \cap B'_1)$ ,  $(A_2 \cap A'_3, B_2 \cap B'_3)$ , and  $(A_3 \cap A'_3, B_3 \cap B'_3)$ , has a perfect matching.

The strong connectivity of  $\mathcal{G}_S$  is a consequence of the following fact.

**Lemma 4.** *There is a path in  $\mathcal{G}_S$  between any pair of perfectly nested matchable splits.*

*Proof.* Let  $X = [(A_1, B_1), (A_2, B_2), (A_3, B_3)]$ ,  $X' = [(A'_1, B'_1), (A'_2, B'_2), (A'_3, B'_3)]$  be two matchable splits, satisfying (N1), (N2) and (N3) above. We Show that there is a path in  $\mathcal{G}_S$  from  $X$  to  $X'$ . A path in the opposite direction can be found similarly. Fix a matching  $M$  in the graph  $(A'_1 \cap A_2, B'_1 \cap B_2)$ . By Proposition 5, there is a vertex  $b \in B'_1 \setminus B_1$ , such that  $b$  is connected by an edge  $\{a, b\}$  to some vertex  $a \in A_1$ . Note that  $b \notin B_3 \cap B'_1$  since the graph  $(A_1, B_3)$  is empty. Thus  $b \in B_2 \cap B'_1$ . Let  $a' \in A'_1 \cap A_2$  be the vertex matched by  $M$  to  $b$ . Now consider the neighbor  $X'' = \mathcal{N}(X, a', b) = [(A''_1, B''_1), (A''_2, B''_2), (A''_3, B''_3)]$  of  $X$  in  $\mathcal{G}_S$ . We claim that  $X''$  is perfectly nested with respect to  $X'$ . To see this

observe, by Proposition 2, that the graph  $(A_2, B_2) - \{a', b\}$  is decomposed, in a unique way, into a matchable graph  $G_2''' = (A_2'', B_2'')$  and a graph with a perfect matching  $G_2'''' = (A_2''', B_2''')$ . On the other hand, the graph  $(A_2', B_2')$  is matchable, while the graph  $(A_2 \cap A_1' \setminus \{a'\}, B_2 \cap B_1' \setminus \{b\}) \cup (A_2 \cap A_3', B_2 \cap B_3')$  has a perfect matching, and therefore by Proposition 3, the union of these two graphs (which is  $(A_2, B_2) - \{a', b\}$ ) decomposes in a unique way into a matchable graph  $F_1$  containing  $(A_2', B_2')$  and a graph with a perfect matching  $F_2$ . The uniqueness of the decomposition implies that  $F_1 = G_2'''$  and  $F_2 = G_2''''$ , i.e.  $A_2'' \supseteq A_2'$  and  $B_2'' \supseteq B_2'$ . Note that the graph  $(A_2'' \cap A_1', B_2'' \cap B_1')$  still has a perfect matching. Note also that the graph  $(A_1'', B_1'')$  is obtained by extending the matchable graph  $(A_1 \cup \{a'\}, B_1 \cup \{b\})$  with pairs from the graph  $(A_2''', B_2''') \cup (A_1' \cap A_3, B_1' \cap B_3)$ , which has a perfect matching. Such an extension must stay within the graph  $(A_1', B_1')$ , i.e.  $A_1'' \subseteq A_1'$  and  $B_1'' \subseteq B_1'$ , since there are no edges between  $A_1'$  and  $B_2''' \cap B_3'$ . Finally, since the extension leaves a perfect matching in the graph  $(A_2'' \setminus A_1'', B_2'' \setminus B_1'') \cup (A_1' \cap A_3 \setminus A_1'', B_1' \cap B_3 \setminus B_1'')$ , we conclude that  $X''$  and  $X'$  are perfectly nested. This way, we obtained a neighbor  $X''$  of  $X$  that is closer to  $X'$  in the sense that  $|A_1''| > |A_1|$ . This implies that there is a path in  $\mathcal{G}_S$  from  $X$  to  $X'$  of length at most  $|A_1' \setminus A_1|$ .  $\square$

**Proof of Lemma 2.** Let  $X = [(A_1, B_1), (A_2, B_2), (A_3, B_3)]$  be a matchable split. Let  $a$  be any vertex in  $A_1$ , then the matchability of  $A_1$  implies that the graph  $(A_1 \setminus \{a\}, B_1)$  has a perfect matching. Thus, with respect to  $a^*$  and  $X$ , the conditions (N1)-(N3) hold, implying that they are perfectly nested. Lemma 4 hence implies that there are paths in  $\mathcal{G}_S$  from  $a^*$  to  $X$  and from  $X$  to  $a^*$ . Similarly we can argue that there are paths in  $\mathcal{G}_S$  between  $X$  and  $b^*$  for any  $b \in B_2$ . In particular, there are paths in  $\mathcal{G}_S$  between  $a^*$  and  $b^*$  for any  $a \in A$  and  $b \in B$ . The lemma follows.  $\square$

## 5 Some Generalizations and Related Problems

### 5.1 $d$ -Factors

Let  $G = (A, B, E)$  be a bipartite graph, and  $d : A \cup B \mapsto \{0, 1, \dots, |A| + |B|\}$  be a non-negative function assigning integer weights to the vertices of  $G$ . We shall assume in what follows that, for each vertex  $v \in A \cup B$ , the degree of  $v$  in  $G$  is at least  $d(v)$ . A  $d$ -factor of  $G$  is a subgraph  $(A, B, X)$  of  $G$  in which each vertex  $v \in A \cup B$  has degree  $d(v)$  (see Section 10 in [16]), i.e., perfect matchings are the 1-factors. Note that  $d$ -factors of  $G$  are in one-to-one correspondence with the vertices of the polytope  $P(G) = \{x \in \mathbb{R}^E \mid Hx = d, \ 0 \leq x \leq 1\}$ , where  $H$  is the incidence matrix of  $G$ . In particular, checking if a bipartite graph has a  $d$ -factor can be done in polynomial time by solving a *capacitated transportation* problem with edge capacities equal to 1, see, e.g., [6].

Since the  $d$ -factors are the vertices of a 0/1-polytope, they can be computed with polynomial delay [5]. We present below a more efficient procedure.

**Theorem 3.** *For any given bipartite graph  $G = (A, B, E)$ , and any non-negative integer function  $d : A \cup B \mapsto \{0, 1, \dots, |A| + |B|\}$ , after computing the first  $d$ -factor, all other  $d$ -factors of  $G$  can be enumerated with delay  $O(|E|)$ .*

*Proof.* We use the flashlight method. Let  $\mathcal{M}_d(G)$  be the set of  $d$ -factors of  $G$ . It is enough to check that condition (F1) is satisfied. Given  $S_1, S_2 \subseteq E$ , we can check in polynomial time whether there is an  $X \in \mathcal{M}_d(G)$  such that  $X \supseteq S_1$  and  $X \cap S_2 = \emptyset$  by checking the existence of a  $d'$ -factor in the graph  $(A, B, E \setminus S_2)$ , where  $d'(v) = d(v) - 1$  if  $v \in A \cup B$  incident to some edge in  $S_1$  and  $d'(v) = d(v)$  for all other vertices  $v \in A \cup B$ .

A more efficient procedure can be obtained as a straightforward generalization of the one in [9]. It is a slight modification of the flashlight method that avoids checking the existence of a  $d$ -factor at each node in the backtracking tree. Given a  $d$ -factor  $X$  in  $G$ , it is possible to find another one, if it exists, by the following procedure. Orient all the edges in  $X$  from  $A$  to  $B$  and all the other edges in  $E \setminus X$  from  $B$  to  $A$ . Then it is not difficult to see that the resulting directed graph  $G'$  has a directed cycle if and only if  $G$  contains another  $d$ -factor  $X' \in \mathcal{M}_d(G)$ . Such an  $X'$  can be found by finding a directed cycle  $C$  in  $G'$ , and taking the symmetric difference between the edges of  $C$  and  $X$ .

Now, the backtracking algorithm proceeds as follows. Each node  $w$  of the backtracking tree is identified with a weight function  $d_w$ , a  $d_w$ -factor  $X_w$ , and a bipartite graph  $G_w = (A, B, E_w)$ . At the root  $r$  of the tree, we compute a  $d$ -factor  $X$  in  $G$ , and let  $d_w \equiv d$  and  $G_w = G$ . At any node  $w$  of the tree, we check if there is another  $d_w$ -factor  $X'_w$  using the above procedure, and if there is none, we define  $w$  to be a leaf. Otherwise, we let  $e_w$  be an arbitrary edge in  $X_w \setminus X'_w$ . This defines the two children  $u$  and  $z$  of  $w$  by:  $d_u(v) = d_w(v) - 1$  for  $v \in e_w$  and  $d_u(v) = d_w(v)$  for all other vertices  $v \in A \cup B$ ,  $X_u = X_w$ ,  $G_u = G_w$ ,  $d_z \equiv d_w$ ,  $X_z = X'_w$ , and finally,  $G_z = G_w - e_w$ . The  $d$ -factors computed at the nodes of the backtracking tree are distinct and they form the complete set of  $d$ -factors of  $G$ .  $\square$

## 5.2 2-Matchings in General Graphs

Let  $G = (V, E)$  be an undirected graph with vertex set  $V$  and edge set  $E$ . Let  $H$  be the incidence matrix of  $G$ . As stated in the introduction, if  $G$  is a bipartite graph then the perfect matchings of  $G$  correspond to the vertices of the polytope  $P(G) = \{x \in \mathbb{R}^E \mid Hx = \mathbf{e}, x \geq 0\}$ . In general graphs the vertices of  $P(G)$  are in one-to-one correspondence with the *basic* perfect 2-matchings of  $G$ , i.e. subsets of edges that form a cover of the vertices with vertex-disjoint edges and vertex-disjoint odd cycles (see [16]). A (not necessarily basic) perfect 2-matching of  $G$  is a subset of edges that covers the vertices of  $G$  with vertex-disjoint edges and vertex-disjoint (even or odd) cycles. Denote respectively by  $\mathcal{M}_2(G)$  and  $\mathcal{M}'_2(G)$  the families of perfect 2-matchings and basic perfect 2-matchings of a graph  $G$ . We show below that, the family  $\mathcal{M}_2(G)$  can be enumerated with polynomial delay, and the family  $\mathcal{M}'_2(G)$  can be enumerated in incremental polynomial time. Theorem 2 will follow from the following two lemmas.

**Lemma 5.** *All perfect 2-matchings of a graph  $G$  can be generated with polynomial delay.*

*Proof.* We use the flashlight method with a slight modification. For  $X \subseteq E$ , let  $\pi(X)$  be the property that the graph  $(V, X)$  has a perfect 2-matching. Then  $\mathcal{F}_\pi = \mathcal{M}_2(G)$ . Define  $\mathcal{F}'_\pi = \{X \subseteq E : \text{the graph } (V, X) \text{ is a vertex-disjoint union of some cycles, some disjoint edges, and possibly one path which maybe of length one}\}$ . Clearly, any  $X \in \mathcal{M}_2(G)$  contains some set  $X' \in \mathcal{F}'_\pi$ . Given  $S_1 \subseteq \mathcal{F}'_\pi$ ,  $S_2 \subseteq E$ , we modify the basic approach described in Section 3.2 in two ways. First, when we consider a new edge  $e \in E \setminus (S_1 \cup S_2)$  to be added to  $S_1$ , we first try an edge incident to the path in  $S_1$ , if one exists. If no such path exists, then any edge  $e \in E \setminus (S_1 \cup S_2)$  can be chosen and defined to be a path of length one in  $S_1 \cup \{e\}$ . Second, when we backtrack, for the first time, on an edge  $e$  defining a path of length one in  $S_1$ , we redefine  $S_1$  by considering  $e$  as a single edge rather than a path of length one. Now it remains to check that (F2) is satisfied. Given  $S_1 \subseteq \mathcal{F}'_\pi$ ,  $S_2 \subseteq E$ , and an edge  $e \in E \setminus (S_1 \cup S_2)$ , chosen as above, such that  $S_1 \cup \{e\} \in \mathcal{F}'_\pi$ , we can check in polynomial time whether there is an  $X \in \mathcal{M}_2(G)$  such that  $X \supseteq S_1 \cup \{e\}$  and  $X \cap S_2 = \emptyset$  in the following way. First, we delete all edges in  $S_2$  from  $G$ . Second, we construct an auxiliary bipartite graph  $G^b$  as follows (see [16]). For every vertex  $v$  of  $G$  we define two vertices  $v'$  and  $v''$  in  $G^b$ , and for every edge  $\{u, v\}$  in  $G$  we define two edges  $\{u'v''\}$  and  $\{u'', v'\}$  in  $G^b$ . Third, we orient all the edges in  $S_1 \cup \{e\}$  in such a way that all cycles and the path (if it exists) become directed. Single edges in  $S_1 \cup \{e\}$  get double orientations. Finally, we mark edges in  $G^b$  corresponding to the oriented arcs in  $X$ : for an arc  $(u, v)$  in  $X$ , we mark the corresponding edge  $\{u', v''\}$  in  $G^b$ . It is easy to see that there is an  $X \in \mathcal{M}_2(G)$  such that  $X \supseteq S_1 \cup \{e\}$  and  $X \cap S_2 = \emptyset$  if and only if there is a perfect matching in  $G^b$  extending the marked edges.  $\square$

**Lemma 6.** *Let  $\mathcal{M}_2(G)$  and  $\mathcal{M}'_2(G)$  be respectively the families of perfect 2-matchings and basic perfect 2-matchings of a graph  $G = (V, E)$ . Then*

$$|\mathcal{M}_2(G)| \leq |\mathcal{M}'_2(G)| + \binom{|\mathcal{M}'_2(G)|}{2}.$$

**Proof of Theorem 2.** By generating all perfect 2-matchings of  $G$  and discarding the non-basic ones, we can get generate all basic perfect 2-matchings. By Lemma 6, the total time for this generation is polynomial in  $|V|$ ,  $|E|$ , and  $|\mathcal{M}'_2(G)|$ . Since the problem is self-reducible, we can convert this output polynomial generation algorithm to an incremental one, see [3,4,8] for more details.  $\square$

**Acknowledgements.** We thank Leonid Khachiyan for helpful discussions.

## References

1. D. Avis, K. Fukuda, A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra, *Symposium on Computational Geometry* 1991, North Conway, NH, pp. 98–104.

2. E. Boros, K. Elbassioni, V. Gurvich and L. Khachiyan, On enumerating minimal dicuts and strongly connected subgraphs, *Integer Programming and Combinatorial Optimization, 10th International IPCO Conference*, (D. Bienstock and G. Nemhauser, eds.) Lecture Notes in Computer Science 3064 (2004) pp. 152–162.
3. E. Boros, K. Elbassioni, V. Gurvich and L. Khachiyan, Algorithms for enumerating circuits in matroids, in *Proc. 14th Annual International Symposium on Algorithms and Computation (ISAAC 2003)*, LNCS 2906, pp. 485–494, Kyoto, Japan, 2003.
4. E. Boros, K. Elbassioni, V. Gurvich and L. Khachiyan, Generating maximal independent sets for hypergraphs with bounded edge-intersections, *6th Latin American Theoretical Informatics Conference (LATIN 2004)*, (Martin Farach-Colton, ed.) Lecture Notes in Computer Science 2461, pp. 424–435.
5. M. Bussieck and M. Lübbecke, The vertex set of a 0/1-polytope is strongly p-enumerable, *Computational Geometry: Theory and Applications* 11(2), pp. 103–109, 1998.
6. W. Cook, W. Cunningham, W. Pulleyblank and A. Schrijver, *Combinatorial Optimization*, John Wiley and Sons, New York, Chapter 4, pp. 91–126.
7. N. Curet, J. DeVinney and M. Gaston, An efficient network flow code for finding all minimum cost  $s$ - $t$  cutsets, *Computers and Operations Research* 29 (2002), pp. 205–219.
8. T. Eiter, G. Gottlob, and K. Makino, New results on monotone dualization and generating hypergraph transversals, *SIAM J. Computing*, 32 (2003) 514–537.
9. K. Fukuda and T. Matsui, Finding all minimum cost perfect matchings in bipartite graphs, *Networks* 22, 1992.
10. D. Gusfield and D. Naor, Extracting maximum information about sets of minimum cuts, *Algorithmica* 10 (1993) pp. 64–89.
11. M. Hall, *Combinatorial theory* (Blaisdell Publ. Co., Waltham, 1967).
12. P.L. Hammer and I.E. Zverovich, Constructing of a maximum stable set with  $k$ -extensions, RUTCOR Research Report RRR 5-2002, Rutgers University, <http://rutcor.rutgers.edu/~rrr/2002.html>.
13. S. Iwata, L. Fleischer, S. Fujishige: A combinatorial, strongly polynomial-time algorithm for minimizing submodular functions, *STOC 2000*, pp. 97–106.
14. E. Lawler, J. K. Lenstra and A. H. G. Rinnooy Kan, Generating all maximal independent sets: NP-hardness and polynomial-time algorithms, *SIAM J. Computing*, 9 (1980) pp. 558–565.
15. L. Lovász, Submodular functions and convexity. In: *Mathematical Programming: The State of the Art*, Bonn 1982, pp. 235–257, (Springer Verlag, 1983).
16. L. Lovász and M. D. Plummer, *Matching theory*, North-Holland, Amsterdam, 1986.
17. K. Murota, *Matrices and Matroids for Systems Analysis*, (Algorithms and Combinatorics, 20), Springer, 1999.
18. J. S. Provan and D. R. Shier, A paradigm for listing  $(s, t)$  cuts in graphs, *Algorithmica* 15, (1996), pp. 351–372.
19. R. C. Read and R. E. Tarjan, Bounds on backtrack algorithms for listing cycles, paths, and spanning trees, *Networks* 5 (1975) 237–252.
20. T. Uno, Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs, *8th International Symposium on Algorithms and Computation, (ISAAC 1997)*, Lecture Notes in Computer Science 1350, pp. 92–101.
21. T. Uno, An Algorithm for Enumerating All Maximal Matchings of a Graph, IPSJ SIGNotes Algorithms Abstract No. 086-007, 2002.

# Direct Routing: Algorithms and Complexity

Costas Busch<sup>1</sup>, Malik Magdon-Ismail<sup>1</sup>, Marios Mavronicolas<sup>2</sup>, and Paul Spirakis<sup>3</sup>

<sup>1</sup> Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180, USA. {buschc,magdon}@cs.rpi.edu

<sup>2</sup> Department of Computer Science, University of Cyprus, P. O. Box 20537, Nicosia CY-1678, Cyprus. mavronic@ucy.ac.cy

<sup>3</sup> Department of Computer Engineering and Informatics, University of Patras, Rion, 265 00 Patras, Greece.

**Abstract.** *Direct routing* is the special case of *bufferless routing* where  $N$  packets, once injected into the network, must be routed along specific paths to their destinations without conflicts. We give a general treatment of three facets of direct routing:

- (i) *Algorithms.* We present a polynomial time *greedy* algorithm for arbitrary direct routing problems which is worst-case optimal, i.e., there exist instances for which no direct routing algorithm is better than the greedy. We apply variants of this algorithm to commonly used network topologies. In particular, we obtain *near-optimal* routing time for the *tree* and *d-dimensional mesh*, given arbitrary sources and destinations; for the *butterfly* and the *hypercube*, the same result holds for random destinations.
- (ii) *Complexity.* By a reduction from Vertex Coloring, we show that Direct Routing is inapproximable, unless  $P=NP$ .
- (iii) *Lower Bounds for Buffering.* We show that certain direct routing problems cannot be solved efficiently; to solve these problems, *any* routing algorithm needs buffers. We give non-trivial lower bounds on such buffering requirements for general routing algorithms.

## 1 Introduction

*Direct routing* is the special case of *bufferless routing* where  $N$  packets are routed along specific paths from *source* to *destination* so that they do not conflict with each other, i.e., once injected, the packets proceed “directly” to their destination without being delayed (buffered) at intermediate nodes. Since direct routing is bufferless, packets spend the minimum possible time in the network given the paths they must follow – this is appealing in power/resource constrained environments (for example optical networks or sensor networks). From the point of view of quality of service, it is often desirable to provide a guarantee on the delivery time after injection, for example in streaming applications like audio and video. Direct routing can provide such guarantees.

The task of a direct routing algorithm is to compute the injection times of the packets so as to minimize the *routing time*, which is the time at which



the last packet is delivered to its destination. Algorithms for direct routing are inherently offline in order to guarantee no conflicts. We give a general analysis of three aspects of direct routing, namely efficient algorithms for direct routing; the computational complexity of direct routing; and, the connection between direct routing and buffering.

*Algorithms for Direct Routing.* We give a general and efficient (polynomial time) algorithm. We modify this algorithm for specific routing problems on commonly used network topologies to obtain more optimal routing times; thus, in many cases, efficient routing can be achieved without the use of buffers.

*Arbitrary:* We give a simple *greedy* algorithm which considers packets in some order, assigning the first available injection time to each packet. This algorithm is worst-case optimal: there exist instances of direct routing for which no direct routing algorithm achieves better routing time.

*Tree:* For arbitrary sources and destinations on arbitrary trees, we give a direct routing algorithm with routing time  $rt = O(rt^*)$  where  $rt^*$  is the minimum possible routing time achievable by *any* routing algorithm (direct or not).

*d-Dimensional Mesh:* For arbitrary sources and destinations on a  $d$ -dimensional mesh with  $n$  nodes, we give a direct routing algorithm with routing time  $rt = O(d^2 \cdot \log^2 n \cdot rt^*)$  with high probability (w.h.p.).

*Butterfly and Hypercube:* For permutation and random destination problems with one packet per node, we obtain routing time  $rt = O(rt^*)$  w.h.p. for the butterfly with  $n$  inputs and the  $n$ -node hypercube.

*Computational Complexity of Direct Routing.* By a reduction from vertex coloring, we show that direct routing is NP-complete. The reduction is *gap-preserving*, so direct routing is as hard to approximate as coloring.

*Lower Bounds for Buffering.* There exist direct routing problems which cannot be efficiently solved; such problems require buffering. We show that for *any buffered algorithm* there exist routing instances, for which packets are buffered  $\Omega(N^{4/3})$  times in order to achieve near-optimal routing time.

Next, we discuss related work, followed by preliminaries and main results.

## Related Work

The only previous known work on direct routing is for permutation problems on trees [1,2], where the authors obtain routing time  $O(n)$  for any tree with  $n$  nodes, which is worst-case optimal. Our algorithm for trees is every-case optimal for arbitrary routing problems. Cypher et al. [3] study an online version of direct routing in which a worm (packet of length  $L$ ) can be re-transmitted if it is dropped (they also allow the links to have bandwidth  $B \geq 1$ ). For the case corresponding to our work ( $L = B = 1$ ), they give an algorithm with routing time  $O((C + \log n) \cdot D)$ . We give an off line algorithm with routing time  $O(C \cdot D)$ , show that this is worst case optimal, and that it is NP-hard to give a good



approximation to the optimal direct routing time. We also obtain near-optimal routing time (with respect to buffered routing) for many interesting networks, for example the Mesh.

A dual to direct routing is time constrained routing where the task is to schedule as many packets as possible within a given time frame [4]. In these papers, the authors show that the time constrained version of the problem is NP-complete, and also study approximation algorithms on linear networks, trees and meshes. They also discuss how much buffering could help in this setting.

Other models of bufferless routing in which packets do not follow specific paths are *matching routing* [5] and *hot-potato routing* [6,7,8]. In [6], the authors also study lower bounds for near-greedy hot-potato algorithms on the mesh. Optimal routing for given paths on arbitrary networks has been studied extensively in the context of store-and-forward algorithms, [9,10,11].

## 2 Preliminaries

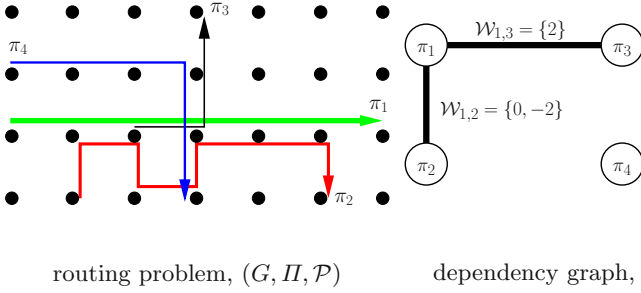
*Problem Definition.* We are given a graph  $G = (V, E)$  with  $n \geq 1$  nodes, and a set of packets  $\Pi = \{\pi_i\}_{i=1}^N$ . Each packet  $\pi_i$  is to be routed from its source,  $s(\pi_i) \in V$ , to its destination,  $\delta(\pi_i) \in V$ , along a pre-specified path  $p_i$ . The nodes in the graph are synchronous: time is discrete and all nodes take steps simultaneously. At each time step, at most one packet can follow a link in each direction (thus, at most two packets can follow a link at the same time, one packet at each direction).

A path  $p$  is a sequence of vertices  $p = (v_1, v_2, \dots, v_k)$ . Two paths  $p_1$  and  $p_2$  *collide* if they share an edge in the same direction. We also say that their respective packets  $\pi_1$  and  $\pi_2$  collide. Two packets *conflict* if they are routed in such a way that they appear in the same node at the same time, *and* the next edge in their paths is the same. The length of a path  $p$ , denoted  $|p|$ , is the number of edges in the path. For any edge  $e = (v_i, v_j) \in p$ , let  $d_p(e)$  denote the length of path  $(v_1, \dots, v_i, v_j)$ . The *distance* between two nodes, is the length of a shortest path that connects the two nodes.

A *direct routing problem* has the following components. **Input:**  $(G, \Pi, \mathcal{P})$ , where  $G$  is a graph, and the packets  $\Pi = \{\pi_i\}_{i=1}^N$  have respective paths  $\mathcal{P} = \{p_i\}_{i=1}^N$ . **Output:** The injection times  $\mathcal{T} = \{\tau_i\}_{i=1}^N$ , denoted a *routing schedule* for the routing problem. **Validity:** If each packet  $\pi_i$  is injected at its corresponding time  $\tau_i$  into its source  $s_i$ , then it will follow a conflict-free path to its destination where it will be absorbed at time  $t_i = \tau_i + |p_i|$ .

For a routing problem  $(G, \Pi, \mathcal{P})$ , the *routing time*  $rt(G, \Pi, \mathcal{P})$  is the maximum time at which a packet gets absorbed at its destination,  $rt(G, \Pi, \mathcal{P}) = \max_i \{\tau_i + |p_i|\}$ . The *offline time*,  $ol(G, \Pi, \mathcal{P})$  is the number of operations used to compute the routing schedule  $\mathcal{T}$ . We measure the efficiency of a direct routing algorithm with respect to the *congestion*  $C$  (the maximum number of packets that use an edge) and the *dilation*  $D$  (the maximum length of any path). A well known lower bound on the routing time of any routing algorithm (direct or not) is given by  $\Omega(C + D)$ .

*Dependency Graphs.* Consider a routing problem  $(G, \Pi, \mathcal{P})$ . The *dependency graph*  $\mathcal{D}$  of the routing problem is a graph in which each packet  $\pi_i \in \Pi$  is a node. We will use  $\pi_i$  to refer to the corresponding node in  $\mathcal{D}$ . There is an edge between two packets in  $\mathcal{D}$  if their paths collide. An edge  $(\pi_i, \pi_j)$  with  $i < j$  in  $\mathcal{D}$  has an associated set of weights  $\mathcal{W}_{i,j}$ :  $w \in \mathcal{W}_{i,j}$  if and only if  $\pi_i$  and  $\pi_j$  collide on some edge  $e$  for which  $d_{\pi_i}(e) - d_{\pi_j}(e) = w$ . Thus, in a valid direct routing schedule with injection times  $\tau_i, \tau_j$  for  $\pi_i, \pi_j$ , it must be that  $\tau_j - \tau_i \notin \mathcal{W}_{i,j}$ . An illustration of a direct routing problem and its corresponding dependency graph are shown in Figure 1.



**Fig. 1.** An example direct routing problem and its dependency graph.

We say that two packets are *synchronized*, if the packets are adjacent in  $\mathcal{D}$  with some edge  $e$  and 0 is in the weight set of  $e$ . A clique  $\mathcal{K}$  in  $\mathcal{D}$  is *synchronized* if all the packets in  $\mathcal{K}$  are synchronized, i.e., if 0 is in the weight set of every edge in  $\mathcal{K}$ . No pair in a synchronized clique can have the same injection time, as otherwise they would conflict. Thus, the size of the maximum synchronized clique in  $\mathcal{D}$  gives a lower bound on the routing time:

**Lemma 1 (Lower Bound on Routing Time).** *Let  $\mathcal{K}$  be a maximum synchronized clique in the dependency graph  $\mathcal{D}$ . Then, for any routing algorithm,  $rt(G, \Pi, \mathcal{P}) \geq |\mathcal{K}|$*

We define the *weight degree* of an edge  $e$  in  $\mathcal{D}$ , denoted  $W(e)$ , as the size of the edge's weight set. We define the *weight degree* of a node  $\pi$  in  $\mathcal{D}$ , denoted  $W(\pi)$ , as the sum of the weight degrees of all edges incident with  $\pi$ . We define the *weight of the dependency graph*,  $W(\mathcal{D})$ , as the sum of the weight degrees of all its edges,  $W(\mathcal{D}) = \sum_{e \in E(\mathcal{D})} W(e)$ . For the example in Figure 1,  $W(\mathcal{D}) = 3$ .

### 3 Algorithms for Direct Routing

Here we consider algorithms for direct routing. The algorithms we consider are variations of the following greedy algorithm, which we apply to the tree, the mesh, the butterfly and the hypercube.

- 1: // **Greedy direct routing algorithm:**
- 2: // **Input:** routing problem  $(G, \Pi, \mathcal{P})$  with  $N$  packets  $\Pi = \{\pi_i\}_{i=1}^N$ .
- 3: // **Output:** Set of injection times  $\mathcal{T} = \{\tau_i\}_{i=1}^N$ .
- 4: Let  $\pi_1, \dots, \pi_N$  be any specific, arbitrarily chosen ordering of the packets.
- 5: **for**  $i = 1$  **to**  $N$  **do**
- 6:   Greedily assign the first available injection time  $\tau_i$  to packet  $\pi_i \in \Pi$ , so that it does not conflict with any packet already assigned an injection time.
- 7: **end for**

The greedy direct routing algorithm is really a family of algorithms, one for each specific ordering of the packets. It is easy to show by induction, that no packet  $\pi_j$  conflicts with any packet  $\pi_i$  with  $i < j$ , and thus the greedy algorithm produces a valid routing schedule. The routing time for the greedy algorithm will be denoted  $rt_{Gr}(G, \Pi, \mathcal{P})$ . Consider the dependency graph  $\mathcal{D}$  for the routing problem  $(G, \Pi, \mathcal{P})$ . We can show that  $\tau_i \leq W(\pi_i)$ , where  $W(\pi_i)$  is the weight degree of packet  $\pi_i$ , which implies:

**Lemma 2.**  $rt_{Gr}(G, \Pi, \mathcal{P}) \leq \max_i \{W(\pi_i) + |p_i|\}$ .

We now give an upper bound on the routing time of the greedy algorithm. Since the congestion is  $C$  and  $|p_i| \leq D \forall i$ , a packet collides with other packets at most  $(C - 1) \cdot D$  times. Thus,  $W(\pi_i) \leq (C - 1) \cdot D, \forall i$ . Therefore, using Lemma 2 we obtain:

**Theorem 1 (Greedy Routing Time Bound).**  $rt_{Gr}(G, \Pi, \mathcal{P}) \leq C \cdot D$ .

The general  $O(C \cdot D)$  bound on the routing time of the greedy algorithm is worst-case optimal, within constant factors, since from Theorem 9, there exist worst-case routing problems with  $\Omega(C \cdot D)$  routing time. In the next sections, we will show how the greedy algorithm can do better for particular routing problems by using a more careful choice of the order in which packets are considered.

Now we discuss the offline time of the greedy algorithm. Each time an edge on a packets path is used by some other packet, the greedy algorithm will need to desynchronize these packets if necessary. This will occur at most  $C \cdot D$  times for a packet, hence, the offline computation time of the greedy algorithm is  $ol_{Gr}(G, \Pi, \mathcal{P}) = O(N \cdot C \cdot D)$ , which is polynomial. This bound is tight since, in the worst case, each packet may have  $C \cdot D$  collisions with other packets.

### 3.1 Trees

Consider the routing problem  $(T, \Pi, \mathcal{P})$ , in which  $T$  is a tree with  $n$  nodes, and all the paths in  $\mathcal{P}$  are shortest paths. Shortest paths are optimal on trees given sources and destinations because any paths must contain the shortest path. Thus,  $rt^* = \Omega(C + D)$ , where  $rt^*$  is the minimum routing time for the given sources and destinations using *any* routing algorithm. The greedy algorithm with a particular order in which the packets are considered gives an asymptotically optimal schedule.

Let  $r$  be an arbitrary node of  $T$ . Let  $d_i$  be the closest distance that  $\pi_i$ 's path comes to  $r$ . The direct routing algorithm can now be simply stated as the greedy algorithm with the packets considered in sorted order, according to the distance  $d_i$ , with  $d_{i_1} \leq d_{i_2} \leq \dots \leq d_{i_N}$ .

**Theorem 2.** *Let  $(T, \Pi, \mathcal{P})$  be any routing problem on the tree  $T$ . Then the routing time of the greedy algorithm using the distance-ordered packets is  $rt(T, \Pi, \mathcal{P}) \leq 2C + D - 2$ .*

*Proof.* We show that every injection time satisfies  $\tau_i \leq 2C - 2$ . When packet  $\pi_i$  with distance  $d_i$  is considered, let  $v_i$  be the closest node to  $r$  on its path. All packets that are already assigned times that could possibly conflict with  $\pi_i$  are those that use the two edges in  $\pi_i$ 's path incident with  $v_i$ , hence there are at most  $2C - 2$  such packets. Since  $\pi_i$  is assigned the smallest available injection time, it must therefore be assigned a time in  $[0, 2C - 2]$ .

### 3.2 $d$ -Dimensional Mesh

A  $d$ -dimensional mesh network  $M = M(m_1, m_2, \dots, m_d)$  is a multi-dimensional grid of nodes with side length  $m_i$  in dimension  $i$ . The number of nodes is  $n = \prod_{i=1}^d m_i$ , and define  $m = \sum_{i=1}^d m_i$ . Every node is connected to up to  $2d$  of its neighbors on the grid. Theorem 1 implies that the greedy routing algorithm achieves asymptotically optimal worst case routing time in the mesh. We discuss some important special cases where the situation is considerably better. In particular, we give a variation of the greedy direct routing algorithm which is analyzed in terms of the number of times that the packet paths “bend” on the mesh. We then apply this algorithm to the 2-dimensional mesh in order to obtain optimal permutation routing, and the  $d$ -dimensional mesh, in order to obtain near-optimal routing, given arbitrary sources and destinations.

*Multi-bend Paths.* Here, we give a variation of the greedy direct routing algorithm which we analyze in terms of the number of times a packet bends in the network. Consider a routing problem  $(G, \Pi, \mathcal{P})$ . We first give an upper bound on the weight degree  $W(\mathcal{D})$  of dependency graph  $\mathcal{D}$  in terms of bends of the paths. We then use the weight degree bound in order to obtain an upper bound on the routing time of the algorithm.

For any subset of packets  $\Pi' \subseteq \Pi$ , let  $\mathcal{D}_{\Pi'}$  denote the subgraph of  $\mathcal{D}$  induced by the set of packets  $\Pi'$ . (Note that  $\mathcal{D} = \mathcal{D}_{\Pi}$ .) Consider the path  $p$  of a packet  $\pi$ . Let's assume that  $p = (\dots, v_i, v, v_j, \dots)$ , such that the edges  $(v_i, v)$  and  $(v, v_j)$  are in different dimensions. We say that the path of packet  $\pi$  *bends* at node  $v$ , and that  $v$  is an *internal bending node*. We define the source and destination nodes of a packet  $\pi$  to be *external bending nodes*. The *segment*  $p' = (v_i, \dots, v_j)$  of a path  $p$ , is a subpath of  $p$  in which only  $v_i$  and  $v_j$  are bending nodes. Consider two packets  $\pi_1$  and  $\pi_2$  whose respective paths  $p_1$  and  $p_2$  collide at some edge  $e$ . Let  $p'_1$  and  $p'_2$  be the two respective segments of  $p_1$  and  $p_2$  which contain  $e$ . Let  $p'$  be the longest subpath of  $p'_1$  and  $p'_2$  which is common to  $p'_1$  and  $p'_2$ ; clearly

$e$  is an edge in  $p'$ . Let's assume that  $p' = (v_i, \dots, v_j)$ . It must be that  $v_i$  is a bending node of one of the two packets, and the same is true of  $v_j$ . Further, none of the other nodes in  $p'$  are bending nodes of either of the two packets. We refer to such a path  $p'$  as a *common subpath*. Note there could be many common subpaths for the packets  $\pi_1$  and  $\pi_2$ , if they meet multiple times on their paths.

Since  $p_1$  and  $p_2$  collide on  $e$ , the edge  $h = (\pi_1, \pi_2)$  will be present in the dependency graph  $\mathcal{D}$  with some weight  $w \in \mathcal{W}_{1,2}$  representing this collision. Weight  $w$  suffices to represent the collision of the two packets on the entire subpath  $p'$ . Therefore, a common subpath contributes at most one to the weight-number of  $\mathcal{D}$ . Let  $A_{\mathcal{P}}$  denote the number of common subpaths. We have that  $W(\mathcal{D}) \leq A_{\mathcal{P}}$ . Therefore, in order to find an upper bound on  $W(\mathcal{D})$ , we only need to find an upper bound on the number of common subpaths.

For each common subpath, one of the packets must bend at the beginning and one at end nodes of the subpath. Thus, a packet contributes to the number of total subpaths only when it bends. Consider a packet  $\pi$  which bends at a node  $v$ . Let  $e_1$  and  $e_2$  be the two edges of the path of  $\pi$  adjacent to  $v$ . On  $e_1$  the packet may meet with at most  $C - 1$  other packets. Thus,  $e_1$  contributes at most  $C - 1$  to the number of common subpaths. Similarly,  $e_2$  contributes at most  $C - 1$  to the number of common subpaths. Thus, each internal bend contributes at most  $2C - 2$  to the number of common subpaths, and each external bend  $C - 1$ . Therefore, for the set of packets  $\Pi'$ , where the maximum number of internal bends is  $b$ ,  $A_{\mathcal{P}} \leq 2(b + 1)|\Pi'|(C - 1)$ . Hence, it follows:

**Lemma 3.** *For any subset  $\Pi' \subseteq \Pi$ ,  $W(\mathcal{D}_{\Pi'}) \leq 2(b + 1)|\Pi'|(C - 1)$ , where  $b$  is the maximum number of internal bending nodes of any path in  $\Pi'$ .*

Since the sum of the node weight degrees is  $2W(\mathcal{D})$ , we have that the average node weight degree of the dependency graph for *any* subset of the packets is upper bounded by  $4(b + 1)(C - 1)$ . We say that a graph  $\mathcal{D}$  is  $K$ -amortized if the average weight degree for every subgraph is at most  $K$ .  $K$ -amortized graphs are similar to balanced graphs [12]. Thus  $\mathcal{D}$  is  $4(b + 1)C$ -amortized. A *generalized coloring* of a graph with weights on each edge is a coloring in which the difference between the colors of adjacent nodes cannot equal a weight.  $K$ -Amortized graphs admit generalized colorings with  $K + 1$  colors. This is the content of the next lemma.

**Lemma 4 (Efficient Coloring of Amortized Graphs).** *Let  $\mathcal{D}$  be a  $K$ -amortized graph. Then  $\mathcal{D}$  has a valid  $K + 1$  generalized coloring.*

A generalized coloring of the dependency graph gives a valid injection schedule with maximum injection time one less than the largest color, since with such an injection schedule no pair of packets is synchronized. Lemma 4 implies that the dependency graph  $\mathcal{D}$  has a valid  $4(b + 1)(C - 1) + 1$  generalized coloring. Lemma 4 essentially determines the order in which the greedy algorithm considers the packets so as to ensure the desired routing time. Hence, we get the following result:

**Theorem 3 (Multi-bend Direct Routing Time).** *Let  $(M, \Pi, \mathcal{P})$  be a direct routing problem on a mesh  $M$  with congestion  $C$  and dilation  $D$ . Suppose that*

each packet has at most  $b$  internal bends. Then there is a direct routing schedule with routing time  $rt \leq 4(b+1)(C-1) + D$ .

*Permutation Routing on the 2-Dimensional Mesh.* Consider a  $\sqrt{n} \times \sqrt{n}$  mesh. In a permutation routing problem every node is the source and destination of exactly one packet. We solve permutation routing problems by using paths with one internal bend. Let  $e$  be a column edge in the up direction. Since at most  $\sqrt{n}$  packet originate and have destination at each row, the congestion at each edge in the row is at most  $O(\sqrt{n})$ . Similarly for edges in rows. Applying Theorem 3, and the fact that  $D = O(\sqrt{n})$ , we then get that  $rt = O(\sqrt{n})$ , which is worst case optimal for permutation routing on the mesh.

*Near Optimal Direct Routing on the Mesh.* Maggs *et al.* [13, Section 3] give a strategy to select paths in the mesh  $M$  for a routing problem with arbitrary sources and destinations. The congestion achieved by the paths is within a logarithmic factor from the optimal, i.e.,  $C = O(dC^* \log n)$  w.h.p., where  $C^*$  is the minimum congestion possible for the given sources and destinations. Following the construction in [13], it can be shown that the packet paths are constructed from  $O(\log n)$  shortest paths between random nodes in the mesh. Hence, the number of bends  $b$  that a packet makes is  $b = O(d \log n)$ , and  $D = O(m \log n)$ , where  $m$  is the sum of the side lengths. We can thus use Theorem 3 to obtain a direct routing schedule with the following properties:

**Theorem 4.** *For any routing problem  $(M, \Pi)$  with given sources and destinations, there exists a direct routing schedule with routing time  $rt = O(d^2 C^* \log^2 n + m \log n)$ , w.h.p..*

Let  $D^*$  denote the maximum length of the shortest paths between sources and destinations for the packets in  $\Pi$ .  $D^*$  is the minimum possible dilation. Let  $rt^*$  denote the optimal routing time (direct or not). For any set of paths,  $C + D = \Omega(C^* + D^*)$ , and so the optimal routing time  $rt^*$  is also  $\Omega(C^* + D^*)$ . If  $D^* = \Omega(m/(d^2 \log n))$ , then  $rt^* = \Omega(C^* + m/(d^2 \log n))$ , so Theorem 4 implies:

**Corollary 1.** *If  $D^* = \Omega(m/(d^2 \log n))$ , then there exists a direct routing schedule with  $rt = O(rt^* d^2 \log^2 n)$ , w.h.p..*

### 3.3 Butterfly and Hypercube

We consider the  $n$ -input butterfly network  $B$ , where  $n = 2^k$ , [14]. There is a unique path from an input node to an output node of length  $\lg n + 1$ . Assume that every input node is the source of one packet and the destinations are randomly chosen.

For packet  $\pi_i$ , we consider the Bernoulli random variables  $x_j$  which are one if packet  $\pi_j$  collides with  $\pi_i$ . Then the degree of  $\pi_i$  in the dependency graph,  $X_i = \sum_j x_j$ . We show that  $E[X_i] = \frac{1}{4}(\lg n - 1)$ , and since the  $x_j$  are independent, we use the Chernoff bound to get a concentration result on  $X_i$ . Thus, we show that w.h.p,  $\max_i X_i = O(\lg n)$ . Since the injection time assigned by the greedy

algorithm to any packet is at most its degree in the dependency graph, we show that the routing time of the greedy algorithm for a random destination problem on the butterfly satisfies  $\mathbf{P}[rt_{Gr}(B, \Pi, \mathcal{P}) \leq \frac{5}{2} \lg n] > 1 - 2\sqrt{2}n^{-\frac{1}{2}}$ . (the details are given in an appendix).

Valiant [15,16] proposed permutation routing on butterfly-like networks by connecting two butterflies, with the outputs of one as the inputs to the other. The permutation is routed by first sending the packets to random destinations on the outputs of the first butterfly. This approach avoids hot-spots and converts the permutation problem to two random destinations problems. Thus, we can apply the result for random destinations twice to obtain the following theorem:

**Theorem 5.** *For permutation routing on the double-butterfly with random intermediate destinations, the routing time of the greedy algorithm satisfies  $\mathbf{P}[rt_{Gr} \leq 5 \lg n] > 1 - 4\sqrt{2}n^{-\frac{1}{2}}$ .*

A similar analysis holds for the hypercube network (see appendix).

**Theorem 6.** *For a permutation routing using random intermediate destinations and bit-fixing paths, the routing time of the greedy algorithm satisfies  $\mathbf{P}[rt_{Gr} < 14 \lg n] > 1 - 1/(16n)$ .*

## 4 Computational Complexity of Direct Routing

In this section, we show that direct routing and approximate versions of it are NP-complete. First, we introduce the formal definition of the direct routing decision problem. In our reductions, we will use the well known NP-complete problem VERTEX COLOR, the vertex coloring problem [17], which asks whether a given graph  $G$  is  $\kappa$ -colorable. The chromatic number,  $\chi(G)$  is the smallest  $\kappa$  for which  $G$  is  $\kappa$ -colorable. An algorithm *approximates*  $\chi(G)$  with *approximation ratio*  $q(G)$  if on any input  $G$ , the algorithm outputs  $u(G)$  such that  $\chi(G) \leq u(G)$  and  $u(G)/\chi(G) \leq q(G)$ . Typically,  $q(G)$  is expressed only as a function of the number of vertices in  $G$ . It is known [18] that unless  $\mathbf{P}=\mathbf{NP}^*$ , there does not exist a polynomial time algorithm to approximate  $\chi(G)$  with approximation ratio  $N^{1/2-\epsilon}$  for any constant  $\epsilon > 0$ .

By polynomially reducing coloring to direct routing, we will obtain hardness and inapproximability results for direct routing. We now formally define a generalization of the direct routing decision problem which allows for collisions. We say that an injection schedule is a valid  $K$ -collision schedule if at most  $K$  collisions occur during the course of the routing (a collision is counted for every collision of every pair of packets on every edge).

**Problem:** APPROXIMATE DIRECT ROUTE

**Input:** A direct routing problem  $(G, \Pi, \mathcal{P})$  and integers  $T, K \geq 0$ ,

**Question:** Does there exist a valid  $k$ -collision direct routing schedule  $\mathcal{T}$  for some  $k \leq K$  and with maximum injection time  $\tau_{max} \leq T$ ?

---

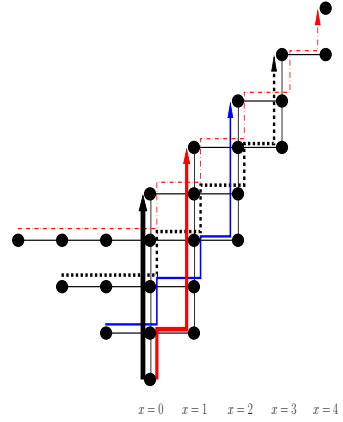
\* It is also known that if  $\mathbf{NP} \not\subseteq \mathbf{ZPP}$  then  $\chi$  is inapproximable to within  $N^{1-\epsilon}$ , however we cannot use this result as it requires both upper and lower bounds.



The problem DIRECT ROUTE is the restriction of APPROXIMATE DIRECT ROUTE to instances where  $K = 0$ . Denoting the maximum injection time of a valid  $K$ -collision injection schedule by  $T$ , we define the  $K$ -collision injection number  $\tau_K(G, \Pi, \mathcal{P})$  for a direct routing problem as the minimum value of  $T$  for which a valid  $K$ -collision schedule exists. We say that a schedule approximates  $\tau_K(G, \Pi, \mathcal{P})$  with ratio  $q$  if it is a schedule with at most  $K$  collisions and the maximum injection time for this schedule approximates  $\tau_K(G, \Pi, \mathcal{P})$  with approximation ratio  $q$ . We now show that direct routing is NP-hard.

**Theorem 7 (DIRECT ROUTE is NP-Hard).** *There exists a polynomial time reduction from any instance  $(G, \kappa)$  of VERTEX COLOR to an instance  $(G', \Pi, \mathcal{P}, T = \kappa - 1)$  of DIRECT ROUTE.*

*Sketch of Proof.* We will use the direct routing problem illustrated to the right, for which the dependency graph is a synchronized clique. Each path is associated to a level, which denotes the  $x$ -coordinate at which the path moves vertically up after making its final left turn. There is a path for every level in  $[0, L]$ , and the total number of packets is  $N = L + 1$ . The level- $i$  path for  $i > 0$  begins at  $(1 - i, i - 1)$  and ends at  $(i, L + i)$ , and is constructed as follows. Beginning at  $(1 - i, i - 1)$ , the path moves right till  $(0, i - 1)$ , then alternating between up and right moves till it reaches level  $i$  at node  $(i, 2i - 1)$  ( $i$  alternating up and right moves), at which point the path moves up to  $(i, L + i)$ . Every packet is synchronized with every other packet, and meets every other packet exactly once.



**Fig. 2.** A mesh routing problem.

Given an instance  $I = (G, K)$  of VERTEX COLOR, we reduce it in polynomial time to an instance  $I' = (G', \Pi, \mathcal{P}, T = K - 1)$  of DIRECT ROUTE. Each node in  $G$  corresponds to a packet in  $\Pi$ . The paths are initially as illustrated in the routing problem above with  $L = N - 1$ . The transformed problem will have dependency graph  $\mathcal{D}$  that is isomorphic to  $G$ , thus a coloring of  $G$  will imply a schedule for  $\mathcal{D}$  and vice versa.

If  $(u, v)$  is not an edge in  $G$ , then we remove that edge in the dependency graph by altering the path of  $u$  and  $v$  without affecting their relationship to any other paths; we do so via altering the edges of  $G'$  by making one of them to pass above the other, thus avoiding the conflict. After this construction, the resulting dependency graph is isomorphic to  $G$ .

DIRECT ROUTE is in NP, as one can check the validity and routing time of a direct routing schedule, by traversing every pair of packets, so DIRECT ROUTE is NP complete. Further, we see that the reduction is gap preserving with gap preserving parameter  $\rho = 1$  [19].



**Theorem 8 (Inapproximability of Collision Injection Number).** *A polynomial time algorithm that approximates  $\tau_K(G, \Pi, \mathcal{P})$  with ratio  $r$  for an arbitrary direct routing problem yields a polynomial time algorithm that approximates the chromatic number of an arbitrary graph with ratio  $r + K + 1$ . In particular, choosing  $K = O(r)$  preserves the approximation ratio.*

For  $K = O(N^{1/2-\epsilon})$ , since  $\chi$  is inapproximable with ratio  $O(N^{1/2-\epsilon})$ , we have

**Corollary 2 (Inapproximability of Scheduling).** *Unless  $P=NP$ , for  $K = O(N^{1/2-\epsilon})$ , there is no polynomial time algorithm to determine a valid  $K$ -collision direct routing schedule that approximates  $\tau_K(G, \Pi, \mathcal{P})$  with ratio  $O(N^{1/2-\epsilon})$  for any  $\epsilon > 0$ .*

## 5 Lower Bounds for Buffering

Here we consider the buffering requirements of any routing algorithm. We construct a “hard” routing problem for which *any* direct routing algorithm has routing time  $rt = \Omega(C \cdot D) = \Omega(C + D)^2$ , which is asymptotically worse than optimal. We then analyze the amount of buffering that would be required to attain near optimal routing time, which results in a lower bound on the amount of buffering needed by any store-and-forward algorithm.

**Theorem 9 (Hard Routing Problem).** *For every direct routing algorithm, there exist routing problems for which the routing time is  $\Omega(C \cdot D) = \Omega((C + D)^2)$*

*Proof.* We construct a routing problem for which the dependency graph is a synchronized clique. The paths are as in Figure 2, and the description of the routing problem is in the proof of Theorem 7. The only difference is that  $c$  packets use each path. The congestion is  $C = 2c$  and the dilation is  $D = 3L$ . Since every pair of packets is synchronized, Lemma 1 implies that  $rt(G, \Pi, \mathcal{P}) \geq N$ . Since  $N = c(L + 1) = \frac{C}{2}(\frac{D}{3} + 1)$ ,  $rt(G, \Pi, \mathcal{P}) = \Omega(C \cdot D)$ . Choosing  $c = \Theta(\sqrt{N})$  and  $L = \Theta(\sqrt{N})$ , we have that  $C + D = \Theta(\sqrt{N})$  so  $C + D = \Theta(\sqrt{C \cdot D})$ .

Let problem  $A$  denote the routing problem in the proof of Theorem 9. We would like to determine how much buffering is necessary in order to decrease the routing time for routing problem  $A$ . Let  $T$  be the maximum injection time (so the routing time is bounded by  $T + D$ ). We give a lower bound on the number of packets that need to be buffered at least once:

**Lemma 5.** *In routing problem  $A$ , if  $T \leq \alpha$ , then at least  $N - \alpha$  packets are buffered at least once.*

*Sketch of Proof.* If  $\beta$  packets are not buffered at all, then they form a synchronized clique, hence  $T \geq \beta$ . Therefore  $\alpha \geq \beta$ , and since  $N - \beta$  packets are buffered at least once, the proof is complete.

If the routing time is  $O(C + D)$ , then  $\alpha = O(C + D)$ . Choosing  $c$  and  $L$  to be  $\Theta(N^{1/2})$ , we have that  $\alpha = O(N^{1/2})$ , and so from Lemma 5, the number of packets buffered is  $\Omega(N)$ :

**Corollary 3.** *There exists a routing problem for which any algorithm will buffer  $\Omega(N)$  packets at least once to achieve asymptotically optimal routing time.*

Repeating problem A appropriately, we can strengthen this corollary to obtain

**Theorem 10 (Buffering-Routing Time Tradeoff).** *There exists a routing problem which, for any  $\epsilon > 0$ , requires  $\Omega(N^{(4-2\epsilon)/3})$  buffering in order to obtain a routing time that is a factor  $O(N^\epsilon)$  from optimal.*

## References

1. Symeonis, A.: Routing on trees. *Information Processing Letters* **57** (1996) 215–223
2. Alstrup, S., Holm, J., de Lichtenberg, K., Thorup, M.: Direct routing on trees. In: *Proc. 9th Symposium on Discrete Algorithms (SODA 98)*. (1998) 342–349
3. Cypher, R., auf der Heide, F.M., Scheideler, C., Vöcking, B.: Universal algorithms for store-and-forward and wormhole routing. (1996) 356–365
4. Adler, M., Khanna, S., Rajaraman, R., Rosen, A.: Time-constrained scheduling of weighted packets on trees and meshes. In: *Proc. 11th Symposium on Parallel Algorithms and Architectures (SPAA)*. (1999)
5. Alon, N., Chung, F., R.L.Graham: Routing permutations on graphs via matching. *SIAM Journal on Discrete Mathematics* **7** (1994) 513–530
6. Ben-Aroya, I., Chinn, D.D., Schuster, A.: A lower bound for nearly minimal adaptive and hot potato algorithms. *Algorithmica* **21** (1998) 347–376
7. Busch, C., Herlihy, M., Wattenhofer, R.: Hard-potato routing. In: *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*. (2000) 278–285
8. Meyer auf der Heide, F., Scheideler, C.: Routing with bounded buffers and hot-potato routing in vertex-symmetric networks. In: *Proc. 3rd European Symposium on Algorithms (ESA)*. (1995) 341–354
9. Leighton, T., Maggs, B., Richa, A.W.: Fast algorithms for finding  $O(\text{congestion} + \text{dilation})$  packet routing schedules. *Combinatorica* **19** (1999) 375–401
10. Meyer auf der Heide, F., Vöcking, B.: Shortest-path routing in arbitrary networks. *Journal of Algorithms* **31** (1999) 105–131
11. Ostrovsky, R., Rabani, Y.: Universal  $O(\text{congestion} + \text{dilation} + \log^{1+\epsilon} N)$  local control packet switching algorithms. In: *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing*, New York (1997) 644–653
12. Bollobás, B.: *Random Graphs*. 2nd edn. Cambridge University Press (2001)
13. Maggs, B.M., auf der Heide, F.M., Vocking, B., Westermann, M.: Exploiting locality for data management in systems of limited bandwidth. In: *IEEE Symposium on Foundations of Computer Science*. (1997) 284–293
14. Leighton, F.T.: *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes*. Morgan Kaufmann, San Mateo (1992)
15. Valiant, L.G.: A scheme for fast parallel communication. *SIAM Journal on Computing* **11** (1982) 350–361
16. Valiant, L.G., Brebner, G.J.: Universal schemes for parallel communication. In: *Proc. 13th Annual ACM Symposium on Theory of Computing*. (1981) 263–277
17. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York (1979)
18. Feige, U., Kilian, J.: Zero knowledge and the chromatic number. In: *IEEE Conference on Computational Complexity*. (1996) 278–287
19. Hochbaum, D.S.: *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, New York (1997)

# Lower Bounds for Embedding into Distributions over Excluded Minor Graph Families

Douglas E. Carroll<sup>1</sup> and Ashish Goel<sup>2\*</sup>

<sup>1</sup> Department of Computer Science, University of California, Los Angeles  
dcarroll@cs.ucla.edu

<sup>2</sup> Departments of Management Science and Engineering and  
(by courtesy) Computer Science, Stanford University  
ashishg@stanford.edu

**Abstract.** It was shown recently by Fakcharoenphol *et al.* [7] that arbitrary finite metrics can be embedded into distributions over tree metrics with distortion  $O(\log n)$ . It is also known that this bound is tight since there are expander graphs which cannot be embedded into distributions over trees with better than  $\Omega(\log n)$  distortion.

We show that this same lower bound holds for embeddings into distributions over any minor excluded family. Given a family of graphs  $F$  which excludes minor  $M$  where  $|M| = k$ , we explicitly construct a family of graphs with treewidth- $(k + 1)$  which cannot be embedded into a distribution over  $F$  with better than  $\Omega(\log n)$  distortion. Thus, while these minor excluded families of graphs are more expressive than trees, they do not provide asymptotically better approximations in general. An important corollary of this is that graphs of treewidth- $k$  cannot be embedded into distributions over graphs of treewidth- $(k - 3)$  with distortion less than  $\Omega(\log n)$ .

We also extend a result of Alon *et al.* [1] by showing that for any  $k$ , planar graphs cannot be embedded into distributions over treewidth- $k$  graphs with better than  $\Omega(\log n)$  distortion.

## 1 Introduction

Many difficult problems can be approximated well when restricted to certain classes of metrics [13,3,4]. Therefore, low distortion embeddings into these restricted classes of metrics become very desirable. We will refer to the original metric which we would like to embed as the *source metric* and the metric into which we would like to embed as the *target metric*.

Tree metrics form one such class of desirable target metrics in the sense that many difficult problems become tractable when restricted to tree metrics. However, they are not sufficiently expressive; i.e. it has been shown that there are classes of metrics which cannot be approximated well by trees. In particular,

---

\* Research supported in part by NSF CAREER award 0133968, and by a Terman Engineering Fellowship.

Rabinovich and Raz [15] have proved that graph metrics cannot be embedded into trees with distortion better than  $\text{girth}/3 - 1$ .

Therefore, subsequent approaches have proposed the use of more expressive classes of metrics. Alon *et al.* [1] showed that any  $n$ -point metric can be embedded with  $2^{O(\sqrt{\log n \log \log n})}$  distortion into distributions over spanning trees. In so doing they demonstrated that such probabilistic metrics are more expressive than tree metrics. Bartal [2] formally defined probabilistic embeddings and proposed using distributions over arbitrary dominating tree metrics. He showed that any finite metric could be embedded into distributions over trees with  $O(\log^2 n)$  distortion. He subsequently improved this bound to  $O(\log n \log \log n)$  [3].

This path culminated recently in the result of Fakcharoenphol *et al.* [7] in which they improved this bound to  $O(\log n)$  distortion. This upper bound is known to be tight since there exist graphs which cannot be embedded into such distributions with better than  $\Omega(\log n)$  distortion. This lower bound follows naturally from the fact that any distribution over trees can be embedded into  $\ell_1$  with constant distortion, and the existence of expander graphs which cannot be embedded into  $\ell_1$  with distortion better than  $\Omega(\log n)$ . Surprisingly, Gupta *et al.* [8] showed that this same bound is in fact achieved by source graph metrics of treewidth-2. Their result is also implicit in the work of Imase and Waxman [10] where they establish a lower bound on the competitive ratio of online Steiner trees.

It is plausible to hypothesize that there are more general and expressive classes of target metrics. We explore using distributions over minor closed families of graphs and show that asymptotically, they are no stronger than distributions over trees. We show a  $\Omega(\log n)$  lower bound on the distortion even when the source metrics are graphs of low treewidth. More precisely, for any minor  $M$ , where  $|M| = k$ , we exhibit a construction for an infinite class of finite metrics of treewidth- $(k + 1)$  for which any embedding into distributions over families of graphs excluding  $M$  achieves  $\Omega(\log n)$  distortion. A corollary of this fact is that treewidth- $k$  graphs cannot be embedded into distributions over treewidth- $(k - 3)$  graphs with distortion less than  $\Omega(\log n)$ .

A weaker result can be inferred directly from Rao [16] who proved that any minor closed family can be embedded into  $\ell_1$  with distortion  $O(\sqrt{\log n})$ . Consequently, the expanders exhibited by Linial *et al.* [13] cannot be embedded into distributions over minor closed families with distortion less than  $\Omega(\sqrt{\log n})$ .

One can derive a lower bound of  $\Omega(\log n)$  by combining the method of Klein *et al.* [11] for decomposing minor excluded graphs with Bartal's [2] proof of the lower bound for probabilistic embeddings into trees. This bound also follows from the recent paper of Rabinovich on the average distortion of embeddings into  $\ell_1$ . However, we show this same bound holds for embeddings of simple, low-treewidth source metrics.

We continue by exploring the difficulty of embedding planar graph metrics into distributions over other minor excluded graph families. Alon *et al.* showed that any embedding of a 2-dimensional grid into a distribution over spanning subtrees must have distortion  $\Omega(\log n)$ . We show that for any fixed  $k$ , the same

lower bound holds for embeddings of 2-dimensional grids into distributions over dominating treewidth- $k$  graph metrics.

Note that our  $\Omega(\log n)$  lower bounds hide polynomial factors of  $k$ .

### 1.1 Techniques

We employ Yao's MiniMax principle to prove both lower bounds – it suffices to show that for some distribution over the edges of the source graph, any embedding of the source graph into a dominating target graph has large expected distortion. In both cases the expected distortion is shown to be logarithmic in the number of vertices in the graph.

For the main result, we show that given any minor  $M$ , one can recursively build a family of source graphs which guarantee a large expected distortion when embedded into a graph which excludes  $M$  as a minor. This structure is detailed in section 3. While we use some elements from the construction of Gupta *et al.* [8], the main building block in our recursive construction is a stretched clique rather than the simpler treewidth-2 graphs used by Gupta *et al.*, and hence, our proof is more involved. As a step in our proof, we show that if graph  $G$  does not contain  $H$  as a minor, then subdivisions of  $H$  can not be embedded into  $G$  with a small distortion.

To show the lower bound for embedding planar graphs into bounded tree-width graphs, we use the notion of nice tree decompositions [5,12] of bounded tree-width graphs. We show that for a range of set sizes, the subsets of a 2-dimensional grid have much larger separators than do their embeddings in the nice tree decompositions; this suffices to prove an  $\Omega(\log n)$  lower bound with a little more work. This argument is presented in section 4.

## 2 Definitions and Preliminaries

Given two metric spaces  $(G, \nu)$  and  $(H, \mu)$  and an embedding  $\Phi : G \rightarrow H$ , we say that the *distortion* of the embedding is  $\|\Phi\| \cdot \|\Phi^{-1}\|$  where

$$\begin{aligned}\|\Phi\| &= \max_{x,y \in G} \frac{\mu(\Phi(x), \Phi(y))}{\nu(x, y)}, \\ \|\Phi^{-1}\| &= \max_{x,y \in G} \frac{\nu(x, y)}{\mu(\Phi(x), \Phi(y))}\end{aligned}$$

and  $(G, \nu)$   $\alpha$ -approximates  $(H, \mu)$  if the *distortion* is no more than  $\alpha$ . We say that  $\mu$  dominates  $\nu$  if  $\mu(\Phi(x), \Phi(y)) \geq \nu(x, y) \forall x, y$ .

**Definition 1.** Given a graph  $G = (V_G, E_G)$ , a tree  $T = (V_T, E_T)$  and a collection  $\{X_i | i \in V_T\}$  of subsets of  $V_G$ , then  $(\{X_i\}, T)$  is said to be a tree decomposition of  $G$  if

1.  $\bigcup_{i \in V_T} X_i = V_G,$

2. for each edge  $e \in E_G$ , there exists  $i \in V_T$  such that the endpoints of  $e$  are in  $X_i$ , and
3. for all  $i, j, k \in V_T$  : if  $j$  lies on the path from  $i$  to  $k$  in  $T$ , then  $X_i \cap X_k \subseteq X_j$ .

The width of a tree decomposition is  $\max_{i \in V_T} |X_i| - 1$ . The treewidth of a graph  $G$  is the minimum width over all tree decompositions of  $G$ .

**Definition 2.** A tree decomposition  $(\{X_i\}, T = (V_T, E_T))$  of a graph  $G = (V_G, E_G)$  is said to be a nice decomposition if:

1.  $T$  is a rooted binary tree
2. if node  $i \in V_T$  has two children  $j_1, j_2$ , then  $X_{j_1} = X_{j_2} = X_i$
3. if node  $i \in V_T$  has one child  $j$ , then either  $X_i \subset X_j$  and  $|X_i| = |X_j| - 1$  or  $X_j \subset X_i$  and  $|X_j| = |X_i| - 1$
4. if node  $i \in V_T$  is a leaf, then  $|X_i| = 1$

**Proposition 1.** [5,12] If graph  $G$  has a tree decomposition of width  $k$ , then  $G$  has a nice tree decomposition of width  $k$ .

The following property follows directly from the definition of a tree decomposition:

**Proposition 2.** For all  $i, j, k \in V_T$  : if  $j$  lies on the path from  $i$  to  $k$  in  $T$ , and  $x_1 \in X_i$  and  $x_2 \in X_k$  then any path connecting  $x_1$  and  $x_2$  in  $G$  must contain a node in  $X_j$ .

For a more complete exposition of treewidth see [12,5].

**Definition 3.** Given a graph  $H$ , a  $k$ -subdivision of  $H$  is defined as the graph resulting from the replacement of each edge by a path of length  $k$ .

We will use  $G_{n,m}$  to denote the  $m$ -subdivision of  $K_n$ . We observe that  $\text{treewidth}(G_{n,m}) = n - 1 = \text{treewidth}(K_n)$ .

### 3 Distributions over Minor Excluded Families

#### 3.1 Outline

Given a family  $F$  which excludes minor  $M$  where  $|M| = k$ , we recursively construct a family of graphs  $H_i$  which embed into any graph in  $F$  with average distortion  $\Omega(\log n)$  where  $n = |H_i|$ . Then by Yao's MiniMax principle, any embedding into a distribution over  $F$  must have distortion  $\Omega(\log n)$  as claimed.

### 3.2 Results

Before proceeding with our main result, we need the following technical lemma<sup>1</sup>.

**Lemma 1.** *Let  $G, H$  be graphs such that  $G$  does not contain  $H$  as a minor and let  $J$  be the  $k$ -subdivision of  $H$ . If  $f$  is an embedding of the metric of  $J$  into the metric of  $G$ , then  $f$  has distortion at least  $k/6 - 1/2$ .*

*Proof.* We will use the linear extension framework of Rabinovich and Raz [15]. While we borrow some of their techniques, we have not found a reduction that would use their lower bounds (on the distortion of an embedding of a graph into another with a smaller Euler characteristic) as a black-box to prove this technical lemma.

The linear extension  $\tilde{Q}$  of a graph  $Q$  is obtained by identifying with each edge of  $Q$  a line of length 1. All the points on all the lines belong to  $\tilde{Q}$ . The metric  $d_Q$  can be extended to the metric  $\tilde{d}_Q$  in the following natural fashion. If  $x$  and  $y$  lie on the same line  $(u, v)$  then  $\tilde{d}_Q(x, y)$  is merely their distance on the line. If  $x$  lies on the line  $(u, v)$  and  $y$  lies on a different line  $(w, r)$ , then

$$\begin{aligned} \tilde{d}_Q(x, y) = \min\{ & \tilde{d}_Q(x, u) + d_Q(u, w) + \tilde{d}_Q(w, y), \\ & \tilde{d}_Q(x, v) + d_Q(v, w) + \tilde{d}_Q(w, y), \\ & \tilde{d}_Q(x, u) + d_Q(u, r) + \tilde{d}_Q(r, y), \\ & \tilde{d}_Q(x, v) + d_Q(v, r) + \tilde{d}_Q(r, y)\} \end{aligned}$$

We will refer to the original vertices as vertices of  $\tilde{Q}$  and to the original edges of  $Q$  as the lines of  $\tilde{Q}$ . We can now extend the embedding  $f$  into a continuous map  $\tilde{f} : \tilde{J} \rightarrow \tilde{G}$  such that

1.  $\tilde{f}$  agrees with  $f$  on vertices of  $\tilde{J}$ , and
2. if  $x \in (u, v)$  where  $(u, v)$  is a line of  $\tilde{J}$  then  $\tilde{f}(x)$  lies on a shortest path from  $\tilde{f}(u)$  to  $\tilde{f}(v)$  in  $\tilde{G}$  such that  $\tilde{d}_J(x, u)/\tilde{d}_J(x, v) = \tilde{d}_G(\tilde{f}(x), \tilde{f}(u))/\tilde{d}_G(\tilde{f}(x), \tilde{f}(v))$ .

Since  $\tilde{f}$  is continuous, the entire line  $(u, v)$  in  $\tilde{J}$  must be mapped to a single shortest path from  $\tilde{f}(u)$  to  $\tilde{f}(v)$  in  $\tilde{G}$ . We will now assume that the distortion  $\alpha$  of  $f$  is less than  $k/6 - 1/2$  and derive a contradiction. But first, we need to state and prove the following useful claim:

*Claim.* If the points  $\tilde{f}(x)$  and  $\tilde{f}(y)$  lie on the same line in  $\tilde{G}$ , the points  $x, x'$  lie on the same line in  $\tilde{J}$ , and the points  $y, y'$  lie on the same line in  $\tilde{J}$ , then  $\tilde{d}_J(x', y') \leq 2\alpha + 1$ .

*Proof.* Suppose  $x, x'$  and  $y, y'$  lie on lines  $(p, q)$  and  $(r, s)$  in  $\tilde{J}$ , respectively. Use  $X$  to denote the quantity  $\tilde{d}_G(\tilde{f}(x), \tilde{f}(y))$  and  $Y$  to denote the quantity

<sup>1</sup> This lemma appears to be folklore. But we have not been able to find a proof in the literature. Our proof is non-trivial and might be useful in other contexts, so we have provided the proof in this paper.

$\tilde{d}_G(\tilde{f}(p), \tilde{f}(x)) + \tilde{d}_G(\tilde{f}(x), \tilde{f}(q)) + \tilde{d}_G(\tilde{f}(r), \tilde{f}(y)) + \tilde{d}_G(\tilde{f}(y), \tilde{f}(s))$ . Since  $\tilde{f}(x)$  and  $\tilde{f}(y)$  lie on the shortest paths from  $\tilde{f}(p)$  to  $\tilde{f}(q)$  and from  $\tilde{f}(r)$  to  $\tilde{f}(s)$ , respectively, we have  $\tilde{d}_G(\tilde{f}(p), \tilde{f}(q)) + \tilde{d}_G(\tilde{f}(r), \tilde{f}(s)) = Y$ . Also, by triangle inequality, we have  $\tilde{d}_G(\tilde{f}(p), \tilde{f}(r)) + \tilde{d}_G(\tilde{f}(p), \tilde{f}(s)) + \tilde{d}_G(\tilde{f}(q), \tilde{f}(r)) + \tilde{d}_G(\tilde{f}(q), \tilde{f}(s)) \leq 4X + 2Y$ . Clearly,  $X \leq 1$  and  $Y \geq 2$ , so we have

$$\frac{\tilde{d}_G(\tilde{f}(p), \tilde{f}(r)) + \tilde{d}_G(\tilde{f}(p), \tilde{f}(s)) + \tilde{d}_G(\tilde{f}(q), \tilde{f}(r)) + \tilde{d}_G(\tilde{f}(q), \tilde{f}(s))}{\tilde{d}_G(\tilde{f}(p), \tilde{f}(q)) + \tilde{d}_G(\tilde{f}(r), \tilde{f}(s))} \leq 4.$$

Since the distortion is  $\alpha$ , we must have

$$\frac{\tilde{d}_J(p, r) + \tilde{d}_J(p, s) + \tilde{d}_J(q, r) + \tilde{d}_J(q, s)}{\tilde{d}_J(p, q) + \tilde{d}_J(r, s)} \leq 4\alpha.$$

But  $\tilde{d}_J(p, q) = \tilde{d}_J(r, s) = 1$ . Also,  $\tilde{d}_J(p, r) + \tilde{d}_J(p, s) + \tilde{d}_J(q, r) + \tilde{d}_J(q, s) \geq 4\tilde{d}_J(x', y') - 4$ . Hence, we have  $(4\tilde{d}_J(x', y') - 4)/2 \leq 4\alpha$ , or  $\tilde{d}_J(x', y') \leq 2\alpha + 1$ .

We will now continue with the proof of Lemma 1. For any edge  $(u, v) \in H$ , consider the  $u$ - $v$  path of length  $k$  in  $\tilde{J}$ . Consider the image of this path in  $\tilde{G}$  under  $\tilde{f}$ . The image need not be a simple path, but must contain a simple path from  $\tilde{f}(u)$  to  $\tilde{f}(v)$ . We choose any such simple path arbitrarily, and call it the representative path for  $(u, v)$ , and denote it by  $P(u, v)$ . Start traversing this path from  $\tilde{f}(u)$  to  $\tilde{f}(v)$  and stop at the first vertex  $q$  which is the image of a point  $x$  on the  $u$ - $v$  path in  $\tilde{J}$  such that  $\tilde{d}_J(u, x) \geq k/2 - \alpha - 1/2$ . Let  $(p, q)$  be the last edge traversed. We will call this the representative line of  $(u, v)$  and denote it by  $L(u, v)$ . Consider a point  $y$  on the  $u$ - $v$  path in  $\tilde{J}$  that maps to  $p$ . The choice of  $p$  and  $q$  implies that  $\tilde{d}_J(u, y) < k/2 - \alpha - 1/2$ , and hence, we can apply claim 1 to conclude that  $k/2 - \alpha - 1/2 \leq d_J(u, x) \leq k/2 + \alpha + 1/2$ . Now, for any point  $z$  not on the  $u$ - $v$  path in  $\tilde{J}$ , we have  $\tilde{d}_J(x, z) \geq k/2 - \alpha - 1/2$ . Since we assumed  $\alpha < k/6 - 1/2$ , we have  $\tilde{d}_J(x, z) > 2\alpha + 1$  and hence,  $\tilde{f}(z)$  can not lie on the line  $L(u, v)$ .

Thus the representative line  $L(u, v)$  has two nice properties: it lies on a simple path from  $\tilde{f}(u)$  to  $\tilde{f}(v)$  and does not contain the image of any point not on the  $u$ - $v$  path in  $\tilde{J}$ .

Now, we perform the following simple process in the graph  $G$ : For every edge  $(u, v)$  of  $H$ , contract all the edges on the representative path  $P(u, v)$  in  $G$  except the representative edge  $L(u, v)$ . Since the representative line of an edge  $(u, v)$  of  $H$  does not intersect the representative path of any other edge of  $H$ , no representative edge gets contracted. The resulting graph is a minor of  $G$ , but also contains  $H$  as a minor, which is a contradiction. Hence  $\alpha \geq k/6 - 1/2$ .

Now we can prove our main result:

**Theorem 1.** *Let  $F$  be a minor closed family of graphs which excludes minor  $M$  where  $|V_M| = k$ . There exists an infinite family of graphs  $H_i$  with treewidth- $(k+1)$  such that any  $\alpha$ -approximation of the metric of  $H_i$  by a distribution over dominating graph metrics in  $F$  has  $\alpha = \Omega(\log |H_i|)$ .*



*Proof.* Assume  $k$  is odd. We proceed by constructing an infinite sequence  $H_i$  of graphs of treewidth- $k$  and show that the minimum distortion with which they can be embedded into a distribution over graph metrics in  $F$  grows without bound as  $i$  increases.

First we construct  $H_1$ : Construct the graph  $H_1$  by taking  $K_k$  and attaching each point  $v$  to source  $s$  with  $(k-1)/2$  disjoint paths of length  $k$  and attaching  $v$  to sink  $t$  with  $(k-1)/2$  paths of length  $k$ . Call  $s$  and  $t$  the terminals of  $H_1$ .

We now show that  $H_1$  has  $m = (2k+1)\binom{k}{2}$  edges and contains  $\binom{k}{2}$  edge-disjoint  $s, t$  paths of length  $2k+1$ . Label the  $s$  to  $v_i$  paths  $sp_{i,1}$  to  $sp_{i,(k-1)/2}$  and the  $v_i$  to  $t$  paths  $tp_{i,1}$  to  $tp_{i,(k-1)/2}$ . Observe that the paths formed as follows are edge disjoint:

path  $sp_{i,j}$  followed by edge  $(v_i, v_{(i+2j) \bmod k})$  followed by path  $tp_{(i+2j) \bmod k, j}$

Note that  $H_1$  has treewidth  $\leq k+1$ .

The  $H_i$  are constructed recursively: For each  $i$ , construct the graph  $H_i$  by replacing every edge in  $H_1$  with a copy of  $H_{i-1}$ . Therefore,  $H_i$  has  $(2k+1)^i \binom{k}{2}^i$  edges and each pair of non-terminal vertices is connected by  $\binom{k}{2}^{i-1}$  edge-disjoint paths of length  $(2k+1)^{i-1}$ . Also note that  $H_i$  has treewidth  $\leq k+1$ .

As in [8] we use Yao's MiniMax Principle to prove the lower bound. We show that there is a distribution  $d$  over the edges of  $H_i$  such that for any embedding of  $H_i$  into a graph which excludes minor  $M$ , an edge chosen randomly from distribution  $d$  has an expected distortion of  $\Omega(i)$ . Then by Yao's MiniMax principle, for any embedding of  $H_i$  into a distribution over graphs in  $F$ , there must be an edge with expected distortion  $\Omega(i)$ . We shall assume a uniform distribution over the edges.

Let  $U$  be a graph which excludes minor  $M$  and let  $\Phi : H_i \rightarrow U$  be an embedding of  $H_i$  into  $U$  such that distances in  $U$  dominate their corresponding distances in  $H_i$ . For each edge  $e \in H_i$  we will give  $e$  the color  $j, 1 \leq j \leq i-1$  if  $\Phi$  distorts  $e$  by at least  $\frac{1}{6}(2k+1)^j - 1/2$ . Note that an edge may have many colors.

Consider the copies of  $G_{k,(2k+1)^{i-1}}$  in  $H_i$ .  $H_i$  contains a copy of  $K_k$  in which every edge has been replaced with a copy of  $H_{i-1}$ . Each copy of  $H_{i-1}$  has  $\binom{k}{2}^{i-1}$  edge disjoint paths of length  $(2k+1)^{i-1}$ . Thus,  $H_i$  clearly contains at least  $\binom{k}{2}^{i-1}$  edge disjoint copies of  $G_{k,(2k+1)^{i-1}}$ .

Clearly, each copy of  $G_{k,(2k+1)^{i-1}}$  contains  $K_k$  and therefore  $M$  as a minor since  $|M| = k$ . Thus, by Lemma 1, at least one edge in each copy of  $G_{k,(2k+1)^{i-1}}$  has distortion  $\geq \frac{1}{6}(2k+1)^{i-1} - 1/2$ . Since  $H_i$  comprises  $m$  copies of  $H_{i-1}$ , it contains  $m\binom{k}{2}^{i-2}$  copies of  $G_{k,(2k+1)^{i-2}}$ . Therefore, there are at least  $m\binom{k}{2}^{i-2}$  edges with color  $i-2$ . In general, there are at least  $\geq m^{i-j-1} \cdot \binom{k}{2}^j$  edges with color  $j$ .

The distortion of an edge is  $\geq \frac{1}{6}(2k+1)^j - 1/2$  where  $j$  is the largest of the edge's colors. For each edge  $e \in E_{H_i}$  let  $C_e$  be the set of colors which apply to

edge  $e$ . Clearly,  $\forall e \in E_{H_i}$ ,

$$\sum_{j \in C_e} \left( \frac{1}{6} (2k+1)^j - 1/2 \right) \leq 2 \cdot \max_{j \in C_e} \left( \frac{1}{6} (2k+1)^j - 1/2 \right)$$

Thus,

$$\begin{aligned} \sum_{e \in E_{H_i}} \max_{j \in C_e} \left( \frac{1}{6} (2k+1)^j - 1/2 \right) &\geq \frac{1}{2} \sum_{e \in E_{H_i}} \sum_{j \in C_e} \frac{1}{7} (2k+1)^j \\ &> \frac{1}{14} \sum_{j=1}^{i-1} |\{e | j \in C_e\}| \cdot (2k+1)^j \\ &\geq \frac{1}{14} \sum_{j=1}^{i-1} m^{i-j-1} \cdot \binom{k}{2}^j \cdot (2k+1)^j \\ &= \frac{1}{14} \sum_{j=1}^{i-1} m^{i-1} \\ &= \frac{1}{14} (i-1) \cdot m^{i-1} \\ &\geq \frac{1}{28m} i \cdot m^i \\ &\geq \frac{1}{28k^3} i \cdot m^i \end{aligned}$$

Then, since the  $H_i$  has  $m^i$  edges, the average distortion is:  $\frac{1}{28k^3} i = \Omega(\log |H_i|)$ .

## 4 Planar Graphs

### 4.1 Outline

First we show that given a 2-dimensional grid, there is a distribution over the edges such that any embedding into a treewidth- $k$  graph metric has high expected distortion. The proof builds on the work of Alon *et al.* [1]. By Yao's Min-Max principle, this is enough to show that the 2-dimensional grid can not be embedded into a distribution over such graphs with distortion less than  $\Omega(\log n)$ .

### 4.2 Results

In this section we will use  $GRID_n$  to denote the planar graph consisting of the  $n \times n$  grid.

**Lemma 2.** (From [1]) *If  $A$  is a set of  $\beta^2$  vertices in  $GRID_n$ , where  $\beta^2 \leq \frac{n^2}{2}$ , then there are at least  $\beta$  rows that  $A$  intersects but does not fill or at least  $\beta$  columns that  $A$  intersects but does not fill.*

**Lemma 3.** (Modified from [1]) If  $A$  is a set of  $\beta^2$  vertices in  $GRID_n$ , where  $\beta^2 \leq \frac{n^2}{2}$ , and  $B$  is a set of at most  $\beta/4$  vertices in  $A$ , then there are at least  $\beta/2$  vertices in  $A$  that have neighbors outside  $A$  and have distance at least  $\frac{\beta}{4|B|}$  from each vertex of  $B$ .

*Proof.* By Lemma 2, there is a set  $C$  of at least  $\beta$  vertices in  $A$  which are in distinct rows or distinct columns and have neighbors outside of  $A$ . Since they are in distinct rows, a vertex of  $B$  can be at distance  $< \frac{\beta}{4|B|}$  of at most  $\frac{\beta}{2|B|}$  vertices in  $C$ . Thus, there are at least  $\frac{\beta}{2}$  vertices at distance at least  $\frac{\beta}{4|B|}$  from each vertex in  $B$ .

**Lemma 4.** Let  $H$  be a graph of treewidth  $k$ ,  $\Phi : GRID_n \rightarrow H$  be an embedding of  $GRID_n$  into  $H$ , and  $\beta \leq n/4$ . Then there are at least  $\frac{n^2}{24\beta}$  edges  $(u, v)$  such that  $d_H(u, v) > \frac{\beta}{16(k+1)}$ .

*Proof.* Since  $H$  has treewidth- $k$ , it must have a tree decomposition of width  $k$ . Moreover, it must have a *nice* tree decomposition  $(X_i, T)$  of width  $k$  by Proposition 1.

Given a subtree  $S$  of  $T$ , define the  $H\text{SIZE}(S) = |\bigcup_{i \in V_S} X_i|$ . Since  $(X_i, T)$  is a nice decomposition, we know that  $T$  has a maximum degree of 3. We also know that for any two adjacent vertices  $i, j \in T$ ,  $|X_i - X_j| \leq 1$ . Thus, every subtree  $S$  of  $T$  has an edge  $e$  such that removing  $e$  creates 2 subtrees each with  $H\text{SIZE}$  at least  $1/3 \cdot H\text{SIZE}(S)$ . Note that if  $S_1$  and  $S_2$  are the two subtrees of  $T$ ,  $\bigcup_{i \in V_{S_1}} X_i$  and  $\bigcup_{i \in V_{S_2}} X_i$  are not disjoint.

Start with  $T$  and successively delete edges from the remaining component with the largest  $H\text{SIZE}$  such that the  $H\text{SIZE}$ s of the resulting subtrees are as evenly divided as possible. Do this until  $\lceil \frac{n^2}{3\beta^2} \rceil - 1$  edges have been deleted and there are  $\lceil \frac{n^2}{3\beta^2} \rceil$  pieces. The smallest piece will always be at least  $1/3$  the  $H\text{SIZE}$  of the previous largest piece. Therefore, on the average these pieces have  $H\text{SIZE} = 3\beta^2$  and the smallest will have  $H\text{SIZE} \geq \beta^2$ .

Since each deleted edge of  $T$  is incident to 2 pieces, the average number of pieces incident with a piece is less than 2. Thus, at least half the pieces are incident with no more than 4 edges.

Each deleted edge in  $T$  represents a set of points which form a vertex cut of  $H$  of size  $\leq k+1$ . Thus, there are  $\frac{n^2}{6\beta^2}$  pieces of  $H\text{SIZE} \geq \beta^2$  which are separated from the rest of  $H$  by a cut of size  $\leq 4(k+1)$ . Let  $A$  be a piece of  $H\text{SIZE} \geq \beta^2$  and let  $B$  be the subset of size  $\leq 4(k+1)$  separating  $A$  from the rest of  $H$ . Then by Lemma 3,  $A$  has at least  $\beta/2$  vertices with neighbors outside of the piece whose distance from the vertices of  $B$  is at least  $\frac{\beta}{16(k+1)}$ . Thus, there are at least  $\frac{n^2}{24\beta}$  edges which are each distorted by a factor of  $\frac{\beta}{16(k+1)}$ .

**Theorem 2.** Any  $\alpha$ -approximation of the metric of  $GRID_n$  by a distribution over dominating treewidth  $k$  graphs has  $\alpha = \Omega(\log n)$ .

*Proof.* Let  $H$  be an arbitrary graph of treewidth  $k$  whose metric dominates that of  $GRID_n$ . By Lemma 4, there are at least  $\frac{n^2}{24\beta}$  edges which are distorted by  $> \beta/16(k+1)$  for any  $\beta \leq \frac{n}{4}$ . Let  $X$  be the distortion of an edge chosen uniformly at random from  $GRID_n$ .  $X$  can only take on non-negative integer values, so the expected distortion is  $E[X] = \sum_{i \geq 1} Prob(X \geq i)$ . For  $i \leq \frac{n}{64(k+1)}$ , let  $\beta = 16(k+1)i$ . Then,

$$\begin{aligned}
 E[X] &= \sum_{i \geq 1} Prob(X \geq i) \\
 &> \sum_{i \geq 1}^{\lfloor n/64(k+1) \rfloor} \frac{n^2}{24 \cdot 16(k+1)i \cdot 2n(n-1)} \\
 &> \sum_{i \geq 1}^{\lfloor n/64(k+1) \rfloor} \frac{1}{2 \cdot 24 \cdot 16(k+1)i} \\
 &= \frac{1}{768(k+1)} \sum_{i \geq 1}^{\lfloor n/64(k+1) \rfloor} \frac{1}{i} \\
 &= \Omega(\log n)
 \end{aligned}$$

Since  $H$  was arbitrarily chosen, then by Yao's MiniMax principle if  $GRID_n$  is embedded into a distribution over treewidth- $k$  graphs, there must be an edge with expected distortion of  $\Omega(\log n)$ .

## 5 Conclusions

It is interesting to note that the inability of minor closed families to approximate all graphs well supports the conjecture [8] [9] that minor closed families of graphs (and therefore distributions over such families) can be embedded into  $\ell_1$  with constant distortion. Since Linial *et al.* [13] showed a lower bound of  $\Omega(\log n)$  for embedding arbitrary graph metrics into  $\ell_1$ , the conjecture further implies that there must be families of graphs which cannot be embedded into distributions over excluded minor graphs with distortion less than  $\Omega(\log n)$ .

However, the particular inability of minor closed families to approximate other minor closed families also eliminates one potential approach to embedding these families into  $\ell_1$ : Gupta *et al.* [8] showed that although treewidth-2 graphs and treewidth-1 graphs are both embeddable into  $\ell_1$  with constant distortion, treewidth-2 graphs are not embeddable into distributions over treewidth-1 graphs with constant distortion. We have shown that a similar structure holds for all higher treewidths. Thus, an approach which attempts to repeatedly embed bounded treewidth graphs into (distributions over) graphs with lower treewidth will not work.

**Acknowledgements.** The authors would like to thank an anonymous reviewer for pointing out the aforementioned folklore result. We would also like to thank Elias Koutsoupias for valuable discussions, and Adam Meyerson and Shailesh Vaya for helpful comments on previous drafts.

## References

1. N. Alon, R. Karp, D. Peleg, and D. West, "A Graph-Theoretic Game and its Application to the k-Server Problem", *SIAM J. Comput.*, 24:1 (1998), pp. 78-100.
2. Y. Bartal, "Probabilistic approximation of metric spaces and its algorithmic applications", In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, 1996, pp. 184-193.
3. Y. Bartal, "On approximating arbitrary metrics by tree metrics", In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, 1998, pp. 161-168.
4. B. Baker, "Approximation Algorithms for NP-complete Problems on Planar Graphs", *J. ACM*, 41 (1994), pp. 153-180.
5. H. Bodlaender, "A partial k-arboretum of graphs with bounded treewidth", *Theoretical Computer Science*, 209 (1998), pp. 1-45.
6. R. Diestel, "Graph Theory", Springer-Verlag, New York, 1997.
7. J. Fakcharoenphol, S. Rao, and K. Talwar, "A Tight Bound on Approximating Arbitrary Metrics by Tree Metrics", In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, 2003, pp. 448-455.
8. A. Gupta, I. Newman, Y. Rabinovich, and A. Sinclair, "Cuts, trees and  $\ell_1$ -embeddings.", In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, 1999, pp. 399-408.
9. C. Chekuri, A. Gupta, I. Newman, Y. Rabinovich, and A. Sinclair, "Embedding k-Outerplanar Graphs into  $\ell_1$ ", In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003, pp. 527-536.
10. M. Imase and B. Waxman, "Dynamic Steiner Tree Problem.", *SIAM J. Discrete Math.*, 4 (1991), pp. 369-384.
11. P. Klein, S. Plotkin, and S. Rao, "Excluded Minors, Network Decomposition, and Multicommodity Flow", In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, 1993, pp. 682-690.
12. T. Kloks, "Treewidth: Computations and Approximations", *Lecture Notes in Computer Science*, Vol. 842, Springer-Verlag, Berlin, 1994.
13. N. Linial, E. London, and Y. Rabinovich, "The geometry of graphs and some of its algorithmic applications", *Combinatorica*, 15 (1995), pp. 215-245.
14. Y. Rabinovich, "On Average Distortion of Embedding Metrics into the Line and into  $\ell_1$ ", In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, 2003, pp. 456-462.
15. Y. Rabinovich and R. Raz, "Lower Bounds on the Distortion of Embedding Finite Metric Spaces in Graphs", *Discrete & Computational Geometry*, 19 (1998), pp. 79-94.
16. S. Rao, "Small distortion and volume preserving embeddings for Planar and Euclidean metrics", In *Proceedings of the 15th Annual Symposium on Computational Geometry*, 1999, pp. 300-306.

# A Parameterized Algorithm for Upward Planarity Testing

## (Extended Abstract)

Hubert Chan\*

School of Computer Science  
University of Waterloo, Canada  
hy3chan@uwaterloo.ca

**Abstract.** Upward planarity testing, or checking whether a directed graph has a drawing in which no edges cross and all edges point upward, is NP-complete. All of the algorithms for upward planarity testing developed previously focused on special classes of graphs. In this paper we develop a parameterized algorithm for upward planarity testing that can be applied to all graphs and runs in  $O(f(k)n^3 + g(k, \ell)n)$  time, where  $n$  is the number of vertices,  $k$  is the number of triconnected components, and  $\ell$  is the number of cutvertices. The functions  $f(k)$  and  $g(k, \ell)$  are defined as  $f(k) = k!8^k$  and  $g(k, \ell) = 2^{3 \cdot 2^\ell} k^{3 \cdot 2^\ell} k!8^k$ . Thus if the number of triconnected components and the number of cutvertices are small, the problem can be solved relatively quickly, even for a large number of vertices. This is the first parameterized algorithm for upward planarity testing.

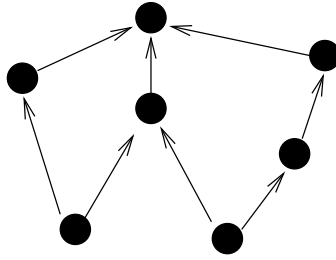
## 1 Introduction

The area of graph drawing deals with geometric representations of abstract graphs, and has applications in many different areas such as software architecture, database design, project management, electronic circuits, and genealogy. These geometrical representations, known as graph drawings, represent each vertex as a point on the plane, and each edge as a curve connecting its two endpoints. Broader treatments of graph drawing are given by Di Battista et al. [6] and by Kaufmann and Wagner [21]. In our discussion of algorithmic results for graphs, we will use the number of vertices,  $n$ , as the input size.

*Upward planarity testing*, or determining whether or not a directed graph can be drawn with no edge crossings such that all edges are drawn upward, is NP-complete [17]. An example of an upward planar drawing is shown in Figure 1. In this paper, we will show how to apply parameterized complexity techniques to solve upward planarity testing. We show how to test upward planarity in  $O(k!8^k n^3 + 2^{3 \cdot 2^\ell} k^{3 \cdot 2^\ell} k!8^k n)$  time, where  $k$  is the number of triconnected components in the graph and  $\ell$  is the number of cutvertices. If the graph is biconnected, we give a  $O(k!8^k n^3)$  time algorithm. Thus if  $k$  is small, we can do upward planarity testing efficiently.

---

\* Research supported by an NSERC Postgraduate Scholarship.



**Fig. 1.** An example of an upward planar drawing

A common approach to solve the problem of upward planarity testing is to look at special classes of graphs. Polynomial-time algorithms have been given for *st*-graphs [9], bipartite graphs [8], triconnected graphs [2], outerplanar graphs [22], and single-source graphs [3,19]. Di Battista and Liotta [7] give a linear time algorithm that checks whether a given drawing is upward planar. Combinatorial characterizations for upward planarity are given by Tamassia and Tollis [25], Di Battista and Tamassia [9], Thomassen [27], and Hutton and Lubiw [19].

For some applications, upward planarity requirements may not be sufficient, or may be too strict. Jünger et al. [20] investigate proper layered drawings, Bertolazzi et al. [1] introduce quasi-upward planarity, and Di Battista et al. [10] investigate the area requirements for upward drawings.

Parameterized complexity, introduced by Downey and Fellows [12] is a technique to develop algorithms that are polynomial in the size of the input, but possibly exponential in one or more parameters, and hence efficient for small fixed parameter values. Although this is the first application of parameterized complexity to upward planarity testing, it has been applied to solve some other problems in graph drawing. Peng [23] investigates applying the concept of treewidth and pathwidth to graph drawing. Parameterized algorithms have been developed for layered graph drawings [13,14], three-dimensional drawings of graphs of bounded path-width [16], the one-sided crossing maximization problem [15], and the two-layer planarization problem [24].

This paper is organized as follows: in Section 2, we define terms that we use in this paper. In Section 3, we develop a parameterized algorithm for upward planarity testing in biconnected graphs. We will then show how to join together biconnected components: given two components  $G_1$  and  $G_2$  to be joined at vertices  $v_1 \in V(G_1)$  and  $v_2 \in V(G_2)$ , we draw one component within a face of the other, draw an edge from  $v_1$  to  $v_2$ , and contract the edge. Thus we must investigate the effects of edge contraction in upward planar graphs (Section 4). We also define a notion of node accessibility in Section 5, which will determine when the edge  $(v_1, v_2)$  can be drawn. In Section 6, we use the results from the previous sections to join together biconnected components, giving us a parameterized algorithm for upward planarity testing in general graphs.

## 2 Definitions

In this paper, we assume the reader is familiar with the standard definitions of graphs and directed graphs, which can be found in a basic graph theory book [4,11]. Given a graph  $G$ , we denote by  $V(G)$  and  $E(G)$  the sets of its vertices and edges, respectively. If there is no confusion as to which graphs we refer, we may simply write  $V$  and  $E$ . Unless otherwise specified, all graphs in this paper are directed simple graphs. In this paper, we adopt the convention of using lowercase Roman letters to represent vertices and lowercase Greek letters to represent edges.

A graph is *biconnected* (*triconnected*) if between any two vertices, there are at least two (three) vertex-disjoint paths. A *cutvertex* is a vertex whose removal disconnects the graph.

Given vertex  $v$  in a directed graph, its *indegree* (*outdegree*), denoted  $\deg^-(v)$  ( $\deg^+(v)$ ), is the number of incoming (outgoing) edges. A vertex with no incoming (outgoing) edges is called a *source* (*sink*).

A *drawing*  $\varphi$  of a graph  $G$  is a mapping of vertices to points on the plane, and edges to curves such that the endpoints of the curve are located at the vertices incident to the edge. In the case of directed graphs, each curve has an associated direction corresponding to the direction of the edge. We say that a curve is *monotone* if every horizontal line intersects the curve at most once. A graph is *planar* if it has a drawing in which no edges cross. A directed graph is *upward planar* if it has a planar drawing in which all edges are monotone and directed upwards.

A planar drawing partitions the plane into regions called *faces*. The infinite, or unbounded, face is called the *outer face*.

A drawing  $\varphi$  defines, for each vertex  $v$ , a clockwise ordering of the edges around  $v$ . The collection  $\Gamma$  of the clockwise orderings of the edges for each vertex is called a (*planar*) *embedding*. If edge  $\epsilon_2$  comes immediately after  $\epsilon_1$  in the clockwise ordering around  $v$ , then we say that  $\epsilon_1$  and  $\epsilon_2$  are *edge-ordering neighbours*,  $\epsilon_2$  is the *clockwise neighbour* of  $\epsilon_1$  around  $v$ , and  $\epsilon_1$  is the *counter-clockwise neighbour* of  $\epsilon_2$ .

In an upward planar drawing, all incoming edges to  $v$  must be consecutive, as must all outgoing edges [25]. Thus for each vertex, we can partition the clockwise ordering of its edges into two linear lists of outgoing and incoming edges. We call the collection of these lists an *upward planar embedding*. If  $v$  is a source (sink), then the first and last edges in the list of outgoing (incoming) edges specify a face. Using the terminology of Bertolazzi et al. [2], we say that  $v$  is a *big angle* on that face.

If a vertex  $v$  has incoming edge  $\epsilon_1$  and outgoing edge  $\epsilon_2$  that are edge ordering neighbours, then we say that the face that contains  $\epsilon_1$  and  $\epsilon_2$  as part of its boundary is *flattenable* at  $v$ .

We *contract* an edge  $\epsilon = (v, w)$  by removing  $v$  and  $w$ , along with their incident edges. We then add a new vertex  $v_\epsilon$ , and for each edge  $(u, v)$  or  $(u, w)$  in  $E(G)$ , we add the edge  $(u, v_\epsilon)$ . For each edge  $(v, u)$  or  $(w, u)$ , we add the edge  $(v_\epsilon, u)$ . The resulting graph is denoted  $G/\epsilon$ . Given an embedding  $\Gamma$  of  $G$ ,



we construct an embedding of  $G/\epsilon$ , denoted  $\Gamma/\epsilon$ , as follows: for each vertex  $u \neq v, w$ , the clockwise ordering of edges around  $u$  remains the same. We then construct the clockwise ordering around  $v_\epsilon$  by first adding the edges incident to  $v$  in order, starting with the clockwise neighbour of  $\epsilon$  around  $v$  and ending with the counter-clockwise neighbour of  $\epsilon$ , then adding the edges incident to  $w$  in order, starting with the clockwise neighbour of  $\epsilon$  around  $w$  and ending with the counter-clockwise neighbour of  $\epsilon$ . We note that  $\Gamma/\epsilon$  may not be an upward planar embedding, even if  $\Gamma$  was; we investigate this further in Section 4.

### 3 Biconnected Components

We first consider only biconnected components; in Section 6, we show how to connect these components together. Our goal for this section is to bound the number of possible planar embeddings of a biconnected graph by a function  $f(k)$ , where  $k$  is the number of triconnected components in the graph, which will allow us to obtain a parameterized algorithm for upward planarity testing.

**Theorem 1.** *Given a planar biconnected graph  $G$  that has  $k$  triconnected components,  $G$  has at most  $k!8^{k-1}$  possible planar embeddings, up to reversal of all the edge orderings.*

*Proof. (outline)* We first show that given two vertices in an embedded triconnected graph, there are at most two faces that contain both vertices. From this, we can show that given two triconnected components  $G_1$  and  $G_2$  of  $G$  that share a common vertex  $v$ , there are at most eight possible embeddings of  $G_1 \cup G_2$ . The eight embeddings come from the fact that there are at most two faces of  $G_1$  in which  $G_2$  can be drawn (which are the two faces that contain both  $v$  and another vertex that lies on a path from  $G_1$  to  $G_2$ ), and vice versa, and that there are two possibilities for the edge orderings of  $G_2$  with respect to  $G_1$ . From this, we can show that if  $G$  has  $k$  triconnected components  $G_1, \dots, G_k$  that all share a common vertex, there are at most  $(k-1)!8^{k-1}$  possible embeddings of  $G_1 \cup \dots \cup G_k$ . Since there are at most  $k$  shared vertices between triconnected components, we have at most  $k(k-1)!8^{k-1} = k!8^{k-1}$  embeddings.  $\square$

Using this bound, we can produce a parameterized algorithm that tests whether  $G$  is upward planar.

**Theorem 2.** *There is an  $O(k!8^k n^3)$ -time algorithm to test whether a biconnected graph is upward planar, where  $n$  is the number of vertices and  $k$  is the number of triconnected components.*

*Proof.* Our algorithm works as follows: first it divides the input graph  $G$  into triconnected components, which can be done in quadratic time [18]. It then tests each possible embedding of  $G$  for upward planarity. By Theorem 1, we have  $k!8^{k-1}$  embeddings. From Euler's formula, we know that for each embedding, there are at most  $n$  possible outer faces. Bertolazzi et al. [2] give a quadratic-time algorithm to determine whether a given embedding and outer face correspond

to an upward planar drawing. Thus we can run the algorithm for each possible embedding and outer face, giving a time complexity of  $O(k!8^kn^3)$ .  $\square$

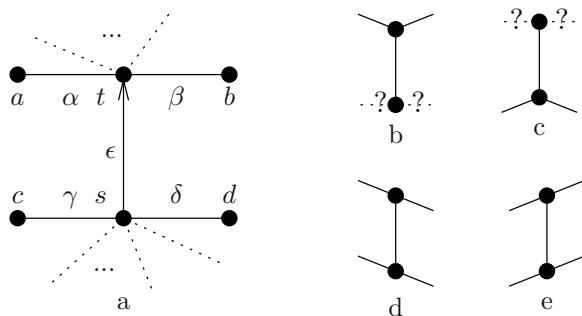
Our bound on the number of possible embeddings is, in many cases, much larger than the actual number of possible embeddings. For example, if each vertex is common to at most two triconnected components, we have only  $8^{k-1}$  possible embeddings, rather than  $k!8^{k-1}$ . We also note that Theorem 1 does not depend on upward planarity. Therefore a similar technique could be applied to other graph drawing problems in which we have an algorithm that solves the problem given a specific embedding.

## 4 Edge Contraction

We now investigate how we can join upward planar embeddings of biconnected graphs in order to obtain an upward planar embedding of a general graph. The joining is achieved by first connecting the biconnected graphs with an edge, and then contracting the edge. Thus we will first examine the effect of edge contraction on upward embeddings. Notably, we will determine conditions under which contraction of an edge in an upward planar embedding produces an embedding that is still upward planar.

Throughout this section, we let  $G$  be an upward planar graph with upward planar embedding  $\Gamma$ , and let  $\epsilon = (s, t)$  be the edge that we wish to contract. We also assume that both  $s$  and  $t$  have degree greater than one; if not, it is easy to see that contracting  $\epsilon$  will result in an upward planar embedding. Thus we will consider the edge-ordering neighbours of  $\epsilon$ . Throughout this section, we will use the following labels. Let  $\alpha = (a, t)$  and  $\beta = (b, t)$  be the clockwise and counterclockwise neighbours of  $\epsilon$  around  $t$  in the embedding  $\Gamma$ , and let  $\gamma = (c, s)$  and  $\delta = (d, s)$  be the counterclockwise and clockwise neighbours of  $\epsilon$  around  $s$  (Figure 2 a).

Since  $\epsilon$  is an arbitrary edge, we must consider the orientations of  $\alpha$ ,  $\beta$ ,  $\gamma$ , and  $\delta$ . As shorthand, if  $\alpha$  or  $\beta$  is oriented towards  $t$ , we say that it is *oriented*



**Fig. 2.** The vertices around the edge  $\epsilon$ , and the edge orientations that we consider.

*inward*, and similarly for when  $\gamma$  or  $\delta$  is oriented towards  $s$ . If  $\alpha$  or  $\beta$  is oriented away from  $t$ , we say that it is *oriented outward*, and similarly for  $\gamma$  and  $\delta$ .

In this paper, we will only consider the four cases for the orientations of  $\alpha$ ,  $\beta$ ,  $\gamma$ , and  $\delta$  that will be used in Section 6, and we only give the lemma statements. The remaining cases, as well as the complete proofs, are given in the thesis from which this work is derived [5].

**Case 1:**  $\alpha$  and  $\beta$  oriented outward,  $\gamma$  and  $\delta$  oriented arbitrarily (Figure 2b),

**Case 2:**  $\gamma$  and  $\delta$  oriented inward,  $\alpha$  and  $\beta$  oriented arbitrarily (Figure 2c),

**Case 3:**  $\alpha$  and  $\gamma$  oriented outward,  $\beta$  and  $\delta$  oriented inward (Figure 2d), and

**Case 4:**  $\alpha$  and  $\gamma$  oriented inward,  $\beta$  and  $\delta$  oriented outward (Figure 2e)

In all four cases, we show that  $G/\epsilon$  is upward planar with embedding  $\Gamma/\epsilon$ . The proof of Lemma 1, which proves Cases 1 and 2, is a straightforward extension of a lemma by Hutton and Lubiw [19], and Lemma 2, which proves Cases 3 and 4, can be easily shown using the characterization given by Bertolazzi et al. [2]. We omit both proofs.

**Lemma 1.** *If  $\deg^-(t) = 1$  ( $\deg^+(s) = 1$ ), then  $G/\epsilon$  is upward planar with upward planar embedding  $\Gamma/\epsilon$ .  $\square$*

**Lemma 2.** *If the edges  $\alpha$  and  $\gamma$  are oriented outward (inward), and  $\beta$  and  $\delta$  are oriented inward (outward), then  $G/\epsilon$  is upward planar with upward planar embedding  $\Gamma/\epsilon$ .  $\square$*

## 5 Node Accessibility

We now define a notion of accessibility of vertices from different parts of the outer face, namely the area above or below the drawing of  $G$ . This, along with the edge contraction results from the previous section, will help us join together upward planar subgraphs.

Given an upward planar graph  $G$  with a specified upward planar embedding  $\Gamma$  and outer face  $F$ , we say that the vertex  $v$  is *accessible from above (below)* if there is an upward planar drawing of  $G$  corresponding to the specified embedding and outer face such that a monotone curve that does not cross any edges can be drawn from  $v$  to a point above (below) the drawing of  $G$ .

Note that if a monotone curve can be drawn from  $v$  to a point  $p$  above the drawing of  $G$ , we can draw a monotone curve from  $v$  to any other point  $q$  above the drawing of  $G$  by appropriately modifying the curve from  $v$  to  $p$ . Therefore our definition of accessibility from above is not dependent on the point to which we draw the curve.

In order to efficiently test whether or not a vertex is accessible from above or from below, we wish to derive conditions for vertex accessibility that we can obtain directly from the planar embedding, rather than from a graph drawing. With the conditions below, we can test whether a vertex  $v$  is accessible from above or from below in  $O(n)$  time.

**Theorem 3.** *If the vertex  $v$  is on the outer face and has an outgoing (incoming) edge  $e$  that is on the outer face, then  $v$  is accessible from above (below).*

*Proof. (outline)* We can first show an equivalent definition of accessibility from above and from below:  $v$  is accessible from above if there is an upward planar drawing of  $G$  such that we can draw a monotone curve from  $v$  to a point  $p$  that is above  $v$  and on the outer face. The point  $p$  need not be above the drawing of  $G$ . Proving that this definition is equivalent is long and largely technical. To prove it, we show how to take a drawing in which we can draw a monotone curve from  $v$  to  $p$ , and modify this drawing so that we can draw a new curve from  $v$  to a new point  $q$  that is above the drawing of  $G$ .

It is then easy to see that if  $v$  has an outgoing edge on the outer face, then we can draw a curve from  $v$  to a point above  $v$  on the outer face.  $\square$

**Theorem 4.** *A source (sink)  $v$  is accessible from below (above) if and only if it is a big angle on the outer face.*

*Proof.* As shown by Bertolazzi et al. [2], in any upward planar drawing of  $G$ , if  $v$  is a big angle on the outer face, then the angle of the outer face formed at  $v$  must be greater than  $180^\circ$ , and hence the outer face must include some area below  $v$ . Thus using the alternate definition of accessibility from the proof of the previous theorem, we can draw a monotone curve from  $v$  to a point below  $v$  on the outer face, and hence  $v$  is accessible from below.

To show the other direction, we note that if  $v$  is a source, there is only one face below  $v$ , and hence  $v$  must be a big angle on that face. Since  $v$  is accessible from below, that face must be the outer face.  $\square$

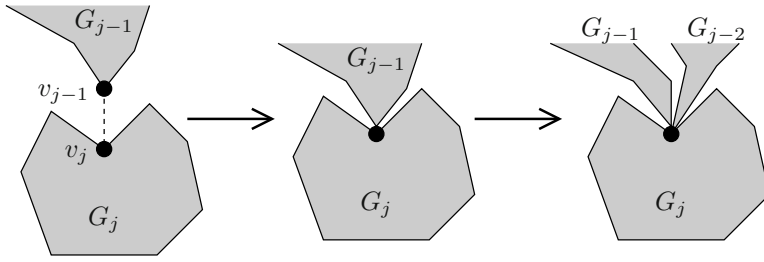
## 6 Joining Biconnected Components

We are now ready to show how to join multiple upward planar biconnected components to form a larger upward planar graph. Throughout this section, we let  $G_1, \dots, G_k$ , with upward planar embeddings  $\Gamma_1, \dots, \Gamma_k$  and outer faces  $F_1, \dots, F_k$ , be connected (not necessarily biconnected) components. We will start by joining the components by identifying the vertices  $v_1 \in V(G_1), \dots, v_k \in V(G_k)$ . The resulting graph we call  $G$ , with upward planar embedding  $\Gamma$ . By repeating this process, we can construct any arbitrary graph, starting from its biconnected components.

To help us prove our conditions for joining graphs, we first show that for all  $i$ , except for at most one,  $v_i$  must be on the outer face of the corresponding  $G_i$  in order for  $G$  to be upward planar.

**Lemma 3.** *If there exist more than one value of  $i$  such that  $G_i$  does not have an upward planar embedding with  $v_i$  on the outer face then  $G$  is not upward planar.*

*Proof. (outline)* We assume that we are given an upward planar drawing of  $G$ , and show that if we consider this drawing, restricted to the subgraph  $G_i$ , then  $v_i$  must be on the outer face for  $i > 1$ .  $\square$



**Fig. 3.** Joining  $G_j$  to  $G_{j-1}$  and then to  $G_{j-2}$  when  $v_j$ ,  $v_{j-1}$ , and  $v_{j-2}$  are all sources.

We identify three cases that we will use to join together components to construct arbitrary graphs.

**Case 1:** All  $v_i$  are sources, or are all sinks.

**Case 2:**  $k = 2$  and  $v_1$  is a source or a sink.

**Case 3:** None of the  $v_i$ 's are sources or sinks, and are not cutvertices.

The way in which we will use these cases is as follows: we will join together all components in which  $v_i$  is a source to form a larger component by using Case 1. Similarly, we join together all components in which  $v_i$  is a sink. Using Case 3, we join together all components in which  $v_i$  is neither a source nor a sink. Finally, using Case 2, we join together the three new components. In each case, we give necessary and sufficient conditions for the upward planarity of  $G$ .

**Theorem 5.** *If  $v_i$  is a source (sink) for all  $i$ , then  $G$  is upward planar if and only if for all  $j > 1$ ,  $G_j$  has an upward planar drawing in which  $v_j$  is on the outer face.*

*Proof.* Since  $v_j$  is on the outer face and is a source, it must have an outgoing edge on the outer face, and so by Theorem 3, must be accessible from above. So we can draw  $G_{j-1}$  in the outer face of  $G_j$  above  $v_j$ , draw an edge from  $v_j$  to  $v_{j-1}$  and contract the edge, using Lemma 1, as illustrated in Figure 3. We start this procedure from the last component, and note that if  $v_j$  and  $v_{j-1}$  are on the outer faces in their respective components, then after  $v_j$  and  $v_{j-1}$  have been identified, the new vertex is still a source on the outer face of the new component.

The other direction follows directly from Lemma 3. □

**Theorem 6.** *Given two components  $G_1$  and  $G_2$  where  $v_1$  is a source (sink),  $G$  is upward planar if and only if  $G_1$  has an upward planar embedding in which  $v_1$  is accessible from below (above), or  $G_2$  has an upward planar embedding in which  $v_2$  is accessible from above (below).*

*Proof. (outline)* We omit the proof that the condition is sufficient as the proof is similar to the proof above. To show the other direction, we assume that  $G$

is upward planar,  $v_1$  is not accessible from below, and  $v_2$  is not accessible from above. We first show that if  $v_1$  and  $v_2$  are not cutvertices of their respective components, then  $G$  is not upward planar. We do this by showing that if  $v_1$  and  $v_2$  are cutvertices, then  $G_1$  is drawn entirely within a single face of  $G_2$ . We then consider the faces of  $G_2$  in which  $G_1$  can be drawn, and show that in every case we obtain a contradiction.

We then show that if  $v_1$  is a cutvertex, we can find a subgraph of  $G_1$  in which  $v_1$  is not a cutvertex and is not accessible from below. Similarly, if  $v_2$  is a cutvertex, we can find a subgraph of  $G_2$  in which  $v_2$  is not a cutvertex and is not accessible from above. From the result above, these two subgraphs cannot be joined to yield an upward planar graph. Since this subgraph of  $G$  is not upward planar,  $G$  cannot be upward planar.  $\square$

**Theorem 7.** *If for all  $i$   $v_i$  has indegree and outdegree at least 1 and  $v_i$  is not a cutvertex, then  $G$  is upward planar if and only if for all  $i > 1$ ,  $G_i$  has an upward planar embedding in which the outer face  $F_i$  is flattenable at  $v_i$ .*

*Proof. (outline)* Again, we omit the proof that the condition is sufficient. We assume that we are only given two components  $G_1$  and  $G_2$  such that  $F_i$  is not flattenable at  $v_i$ : if we have more than two components such that  $F_i$  is not flattenable at  $v_i$  for more than one value of  $i$ , we can take any two such components as  $G_1$  and  $G_2$  and show that  $G$  is not upward planar. Again, since  $v_1$  and  $v_2$  are not cutvertices, in any upward planar drawing of  $G$ ,  $G_1$  must be drawn entirely within a single face of  $G_2$  and vice versa, and in all cases, we obtain a contradiction and hence  $G$  cannot be upward planar.  $\square$

We can now give a fixed-parameter algorithm for determining when a graph is upward planar, with the parameter being the number of triconnected components and the number of cutvertices.

**Theorem 8.** *There is an  $O(k!8^kn^3 + 2^{3 \cdot 2^\ell} k^{3 \cdot 2^\ell} k!8^kn)$ -time algorithm to test whether a graph  $G$  is upward planar, where  $n$  is the number of vertices,  $k$  is the number of triconnected components, and  $\ell$  is the number of cutvertices.*

*Proof. (outline)* Our algorithm works by splitting  $G$  into biconnected components, generating all possible upward planar embeddings for the biconnected components, and joining them together, keeping track of all possible upward planar embeddings for the new components.

We can  $G$  split into biconnected components in  $O(n)$  time [26]. For each biconnected component, we generate all possible upward planar embeddings, along with their possible outer faces, as shown in Section 3. In total, this takes  $O(k!8^kn^3)$  time, and generates at most  $k!8^{k-1}$  embeddings.

For each cutvertex  $v$ , we first join together all components in which  $v$  has indegree and outdegree at least one by using Theorem 7, producing the new component  $G_\times$ . We then use Theorem 5 to join together all components in which  $v$  is a source, producing  $G_\vee$ , and all components in which  $v$  is a sink, producing  $G_\wedge$ . Then, using Theorem 6, we join  $G_\vee$  to  $G_\times$  and join  $G_\wedge$  to the resulting

component, producing a new component  $G_v$ . We remove all the components that contained  $v$ , replace them with  $G_v$ , and continue inductively. In this step, we may also detect that no possible upward planar embedding exists.

Since the conditions given by the theorems in this section only depend on the accessibility of vertices or on vertices being on the outer face, we can greatly limit the number of embeddings that we must consider. For example, if  $G_i$  is not on the outer face (and hence not accessible from above or from below) in the embeddings  $I_1$  and  $I_2$  of  $G_v$  and we later join  $G_v$  to a component that shares a vertex with  $G_i$ ,  $I_1$  can be used to create an upward planar embedding of the new graph if and only if  $I_2$  can also be used. Thus we only need to consider either  $I_1$  or  $I_2$ .

We can show that, for the  $i$ th cutvertex, the number of embeddings that we will produce will be less than  $2^{2^i} k^{2^i} k! 8^k$ , and producing them will take at most  $O(2^{3 \cdot 2^i} k^{3 \cdot 2^i} k! 8^k n)$  time. Since we have  $\ell$  cutvertices, summing over all the steps for joining biconnected components gives a time of at most  $O(2^{3 \cdot 2^\ell} k^{3 \cdot 2^\ell} k! 8^k n)$ .

Thus in total, the algorithm runs in  $O(k! 8^k n^3 + 2^{3 \cdot 2^\ell} k^{3 \cdot 2^\ell} k! 8^k n)$  time.  $\square$

## 7 Conclusions and Future Work

In this paper, we first developed a parameterized algorithm for upward planarity testing of biconnected graphs. This algorithm runs in  $O(k! 8^k n^3)$  time, where  $k$  is the number of triconnected components. We then showed conditions under which contracting an edge in an upward planar graph results in a new graph that is still upward planar, and we introduced a notion of vertex accessibility from above and below. Using these results, we then gave necessary and sufficient conditions for joining biconnected graphs to form a new upward planar graph. This allowed us to obtain a parameterized algorithm for upward planarity testing in general graphs. Our algorithm runs in  $O(k! 8^k n^3 + 2^{3 \cdot 2^\ell} k^{3 \cdot 2^\ell} k! 8^k n)$  time, where  $n$  is the number of vertices,  $k$  is the number of triconnected components, and  $\ell$  is the number of cutvertices.

Our running time analysis contains many potential overestimations. Better analysis may yield a smaller running time. It would also be interesting to implement the algorithm and see how well it performs in practice. In particular, since the complexity analysis includes bookkeeping of embeddings that are not upward planar, it is very likely that the running time in practice will be much smaller than that given in Theorem 8.

Another possible research direction in applying parameterized complexity to upward planarity testing is obtaining a parameterized algorithm that determines whether or not a graph has a drawing in which at most  $\frac{1}{k}$  of the edges point downward. For  $k = \infty$ , this is simply upward planarity testing. For  $k = 2$ , this is trivial: take any drawing of the graph in which no edge is drawn horizontally. Either this drawing, or the drawing that results from flipping it vertically, has at least half the edges pointing upward. Thus it is possible that between these two extremes, we may be able to obtain a parameterized result.

Parameterized complexity is a fairly new area and seeks to find efficient solutions to hard problems. Many problems in graph drawing have been shown to be NP-complete, and so parameterized complexity may be able to offer solutions to many of these problems. Some possible parameters worth examining are the height, width, or area of the drawing, the maximum indegree or outdegree in a graph, the number of faces in a planar graph, or the number of sources or sinks.

**Acknowledgements.** I gratefully acknowledge the suggestions of the anonymous referees.

## References

1. Paola Bertolazzi, Giuseppe Di Battista, and Walter Didimo. Quasi-upward planarity. *Algorithmica*, 32:474–506, 2002.
2. Paola Bertolazzi, Giuseppe Di Battista, Giuseppe Liotta, and Carlo Mannino. Upward drawings of triconnected digraphs. *Algorithmica*, 12:476–497, 1994.
3. Paola Bertolazzi, Giuseppe Di Battista, Carlo Mannino, and Roberto Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM Journal on Computing*, 27(1):132–196, 1998.
4. John Adrian Bondy and U. S. R. Murty. *Graph Theory with Applications*. North Holland, New York, 1976.
5. Hubert Chan. A parameterized algorithm for upward planarity testing of biconnected graphs. Master’s thesis, School of Computer Science, University of Waterloo, May 2003.
6. Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, New Jersey, 1999.
7. Giuseppe Di Battista and Giuseppe Liotta. Upward planarity checking: “Faces are more than polygons” (extended abstract). In Sue H. Whitesides, editor, *Proceedings of Graph Drawing 1998: 6th International Symposium*, volume 1547 of *Lecture Notes in Computer Science*, pages 72–86. Springer-Verlag, 1998.
8. Giuseppe Di Battista, Wei-Ping Liu, and Ivan Rival. Bipartite graphs, upward drawings, and planarity. *Information Processing Letters*, 36:317–322, 1990.
9. Giuseppe Di Battista and Roberto Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoretical Computer Science*, 61:175–198, 1988.
10. Giuseppe Di Battista, Roberto Tamassia, and Ioannis G. Tollis. Area requirement and symmetry display of planar upward drawings. *Discrete and Computational Geometry*, 7:381–401, 1992.
11. Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, New York, 2nd edition, 2000.
12. Rod G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, New York, 1997.
13. Vida Dujmović, Michael R. Fellows, Michael T. Hallett, Matthew Kitching, Giuseppe Liotta, Catherine McCartin, Naomi Nishimura, Prabhakar Ragde, Frances Rosamond, Matthew Suderman, Sue Whitesides, and David R. Wood. A fixed-parameter approach to two-layer planarization. In *Proceedings of the 9th International Symposium on Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 1–15, 2001.



14. Vida Dujmović, Michael R. Fellows, Michael T. Hallett, Matthew Kitching, Giuseppe Liotta, Catherine McCartin, Naomi Nishimura, Prabhakar Ragde, Frances Rosamond, Matthew Suderman, Sue Whitesides, and David R. Wood. On the parameterized complexity of layered graph drawing. In *Proceedings, 9th Annual European Symposium on Algorithms*, pages 488–499, 2001.
15. Vida Dujmović, Henning Fernau, and Michael Kaufmann. Fixed parameter algorithms for ONE-SIDED CROSSING minimization revisited. In Giuseppe Liotta, editor, *Proceedings of Graph Drawing 2003: 11th International Symposium*, volume 2912 of *Lecture Notes in Computer Science*. Springer, 2004.
16. Vida Dujmović, Pat Morin, and David R. Wood. Path-width and three-dimensional straight-line grid drawings of graphs. In Stephen G. Kobourov and Michael T. Goodrich, editors, *Proceedings of Graph Drawing 2002*, volume 2528 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2002.
17. Ashim Garg and Roberto Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM Journal on Computing*, 31(2):601–625, 2001.
18. John Hopcroft and Robert E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2:136–158, 1972.
19. Michael D. Hutton and Anna Lubiw. Upward planar drawing of single source acyclic digraphs. *SIAM Journal on Computing*, 25(2):291–311, 1996.
20. Michael Jünger, Sebastian Leipert, and Petra Mutzel. Level planarity testing in linear time. In S. Whitesides, editor, *Proceedings of Graph Drawing 1998: 6th International Symposium*, volume 1547 of *Lecture Notes in Computer Science*, pages 224–237. Springer, 1998.
21. Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science Tutorial*. Springer, 2001.
22. Achilleas Papakostas. Upward planarity testing of outerplanar dags (extended abstract). In Roberto Tamassia and Ioannis G. Tollis, editors, *Proceedings of Graph Drawing 1994*, volume 894 of *Lecture Notes in Computer Science*, pages 298–306. Springer, 1995.
23. Zhou Peng. Drawing graphs of bounded treewidth/pathwidth. Master’s thesis, Department of Computer Science, University of Auckland, 2001.
24. Matthew Suderman and Sue Whitesides. Experiments with the fixed-parameter approach for two-layer planarization. In Giuseppe Liotta, editor, *Proceedings of Graph Drawing 2003: 11th International Symposium*, volume 2912 of *Lecture Notes in Computer Science*. Springer, 2004.
25. Roberto Tamassia and Ioannis G. Tollis. A unified approach to visibility representations of planar graphs. *Discrete and Computational Geometry*, 1(4):321–341, 1986.
26. Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):145–159, 1972.
27. Carsten Thomassen. Planar acyclic oriented graphs. *Order*, 5:349–361, 1989.

# Fisher Equilibrium Price with a Class of Concave Utility Functions

Ning Chen<sup>1</sup>, Xiaotie Deng<sup>2\*</sup>, Xiaoming Sun<sup>3\*\*</sup>, and Andrew Chi-Chih Yao<sup>4\*\*\*</sup>

<sup>1</sup> Department of Computer Science, Fudan University, China  
nchen@fudan.edu.cn

<sup>2</sup> Department of Computer Science, City University of Hong Kong  
csdeng@cityu.edu.hk

<sup>3</sup> Department of Computer Science and Technology, Tsinghua University, China  
sun\_xm97@mails.tsinghua.edu.cn

<sup>4</sup> Department of Computer Science, Princeton University  
yao@cs.princeton.edu

**Abstract.** In this paper we study efficient algorithms for computing equilibrium price in the Fisher model for a class of nonlinear concave utility functions, the logarithmic utility functions. We derive a duality relation between buyers and sellers under such utility functions, and use it to design a polynomial time algorithm for calculating equilibrium price, for the special case when either the number of sellers or the number of buyers is bounded by a constant.

## 1 Introduction

Equilibrium price is a vital notion in classical economic theory, and has been a central issue in computational economics. For the Fisher model, there are  $n$  buyers each with an initially endowed amount of cash and with a non-decreasing concave utility function, and there are  $m$  goods (w.l.o.g., each of a unit amount) for sale. At the equilibrium price, all goods are sold, all cash are spent, and the goods purchased by each buyer maximizes its utility function for the equilibrium price vector as constrained by its initial endowment. It can be viewed as a special case of the more widely known model of Arrow-Debreu.

Arrow and Debreu [1] proved the existence of equilibrium price assuming goods are divisible. Their proof was based on the fixed point theorem and thus was not constructive. Gale made an extensive study for linear utility functions [7]. On the algorithmic side, Scarf's pioneering work [11] pointed to the possibility

---

\* Research work presented in this work is partially supported by an SRG grant of City University of Hong Kong (No. 7001514) and a research grant (CityU 1081/02E) from Hong Kong Research Grants Council, China.

\*\* Research is supported by research grants (60223004, 60321002) of National Natural Science Foundation of China.

\*\*\* The research of this author was supported in part by US National Science Foundation under grant CCR-0310466

of obtaining the equilibrium price in the limit through an iterative combinatorial algorithm. However, after Scarf's work the algorithmic study in computation of equilibrium price has for a long time seen no significant progress.

Recently, Deng, Papadimitriou, and Safra [3] started a study of algorithmic complexity for the equilibrium price problem in the *Arrow-Debreu model*. Complexity and algorithmic results for the equilibrium price are obtained. In addition, an algorithmic concept of approximate equilibrium was introduced that led to a polynomial time approximation scheme for the equilibrium price of an economy with a bounded number of indivisible goods. Even though the work was presented for linear utility functions, it was pointed out many of the results hold for general concave utility functions. Still, a crucial open question remains: is there a polynomial time algorithm for the equilibrium price when the number of goods and agents are part of the input size?

Subsequently, a fair number of explorative works have followed for equilibrium and approximate equilibrium price under linear utility functions to study the problem with general input size. Devanur, et al. [4], obtained a polynomial time algorithm for a price equilibrium for the *Fisher model*, in which agents are initialed with certain amount of money. Later, on the basis of the duality type of algorithm proposed in [4], Jain et al. [8], Devanur and Vazirani [5], and Garg and Kapoor [9] showed different polynomial time approximation schemes for the linear utility functions in the Fisher model. Unfortunately, those results were for the linear utility functions, which still leaves open a gap towards the general concave utility functions.

In this paper, we develop a solution for a class of concave utility functions, the logarithmic utility functions, by making use of an interesting duality relationship between the amount of cash of buyers and the amount of commodities of the sellers. Polynomial time algorithms are obtained when the number of sellers (or buyers) is bounded by a constant.

In Section 2, we introduce a formal formulation of the problem. In Section 3, we derive and discuss a duality relationship between buyers and sellers. In Section 4, we present our polynomial time algorithms. We conclude our work with remarks and discussion in Section 5.

## 2 A Fisher Equilibrium Problem

We study a market consisting of a set  $A$  of  $n$  buyers and a set  $B$  of  $m$  divisible goods (*i.e.*,  $|A| = n$  and  $|B| = m$ ). Let  $e_i$  be the initial amount of endowment of buyer  $i$ ,  $q_j$  be the quantity of goods  $j$ . Though it is common to scale quantities of goods to the units, for clarity we keep the notation  $q_j$ 's for the goods, especially because of the duality relationship we are going to obtain.

We consider a class of logarithmic utility functions:

$$u_i(x_{i,1}, \dots, x_{i,m}) = \sum_{j=1}^m a_{i,j} \ln(x_{i,j}/\epsilon_{i,j} + 1),$$

for each buyer  $i$ , where  $a_{i,j}, \epsilon_{i,j} > 0$  are given constants. For each  $i, j$ , define the functions  $u_{i,j}^{(1)}(t) = \frac{a_{i,j}}{t + \epsilon_{i,j}}$ . Clearly, the first order derivative of the utility function  $u_i$  with respect to goods  $j$  depends only on one variable  $x_{i,j}$ , and is in fact equal to  $u_{i,j}^{(1)}(x_{i,j})$ .

The functions  $u_{i,j}^{(1)}(t)$  are non-negative and strictly decreasing in  $t$  (for  $t > -\epsilon_{i,j}$ ). It represents the marginal utility of buyer  $i$  when the amount of  $j$  held by  $i$  is  $t$ ,  $0 \leq t \leq q_j$ . In comparison, linear utility functions correspond to the case the marginal utility functions,  $u_{i,j}^{(1)}(t)$ , are all constant.

Given a price vector  $P = (p_1, \dots, p_m) > 0$ , each buyer  $i$  has a unique *optimal allocation*  $(x_{i,1}^*(P), \dots, x_{i,m}^*(P))$  (see, e.g., [10]), that maximizes the utility function  $u_i$  of buyer  $i$ :

$$\begin{aligned} \max \quad & u_i(x_{i,1}, \dots, x_{i,m}) \\ \text{s.t.} \quad & p_1 x_{i,1} + \dots + p_m x_{i,m} = e_i \\ & x_{i,j} \geq 0, \quad \forall 1 \leq j \leq m \end{aligned} \quad (1)$$

The *Fisher equilibrium* problem is to find a price vector  $P = (p_1, \dots, p_m)$  (and the corresponding optimal allocations  $x^*(P)$ ) such that all money are spent and all goods are *cleared*, i.e.,

$$\begin{cases} \forall i \in A, \sum_{j \in B} p_j x_{i,j}^*(P) = e_i \\ \forall j \in B, \sum_{i \in A} x_{i,j}^*(P) = q_j \end{cases}.$$

Such a price vector  $P$  is called a (*market*) *equilibrium price vector*.

We relax the last constraint of (1) to obtain a relaxed linear program. For each buyer  $i$ ,

$$\begin{aligned} \max \quad & u_i(x_{i,1}, \dots, x_{i,m}) \\ \text{s.t.} \quad & p_1 x_{i,1} + \dots + p_m x_{i,m} = e_i, x_{i,j} > -\epsilon_{i,j} \text{ for all } j. \end{aligned} \quad (2)$$

Note that the value of utility  $u_i$  approaches  $-\infty$ , as any  $x_{i,j}$  approaches  $-\epsilon_{i,j}$ . Furthermore, the feasible region of  $(x_{i,1}, x_{i,2}, \dots, x_{i,m})$  in  $R^m$  is bounded. It is clear that the maximum of  $u_i$  is achieved in some interior point of the region, and can be found by applying the Lagrange Multiplier Method. Let

$$F(x_{i,1}, \dots, x_{i,m}, \eta) = u_i(x_{i,1}, \dots, x_{i,m}) - \eta(p_1 x_{i,1} + \dots + p_m x_{i,m} - e_i).$$

Setting the derivatives to zero, we obtain

$$\begin{aligned} F_\eta &= e_i - (p_1 x_{i,1} + \dots + p_m x_{i,m}) = 0 \\ F_{x_{i,j}} &= \frac{a_{i,j}}{x_{i,j} + \epsilon_{i,j}} - \eta p_m = 0, j = 1, 2, \dots, m. \end{aligned}$$

The solution  $(x'_{i,1}(P), \dots, x'_{i,m}(P), \frac{1}{\eta})$  is then:

$$\frac{1}{\eta} = \frac{e_i + \sum_{j=1}^m \epsilon_{i,j} p_j}{\sum_{j=1}^m a_{i,j}}$$

$$x'_{i,j} = \frac{a_{i,j}}{p_j} * \frac{e_i + \sum_{j=1}^m \epsilon_{i,j} p_j}{\sum_{j=1}^m a_{i,j}} - \epsilon_{i,j}, j = 1, 2, \dots, m.$$

Note that  $x'_{i,j} > -\epsilon_{i,j}$ , and thus  $(x'_{i,1}(P), \dots, x'_{i,m}(P))$  is indeed an interior point within the domain within which the utility function  $u_i$  is defined (as a real-valued function).

Recall that we use  $(x^*_{i,1}(P), \dots, x^*_{i,m}(P))$  to denote the optimal solution of (1). In addition, in case of no ambiguity, we drop  $P$  in the notation of  $x'$  and  $x^*$ .

**Proposition 1.** *For any  $i, j$ , if  $x'_{i,j} \leq 0$ , then  $x^*_{i,j} = 0$ .*

*Proof.* Let  $i$  be fixed. Assume to the contrary that there exists  $j_1 \in B$  such that  $x'_{i,j_1} \leq 0$  and  $x^*_{i,j_1} > 0$ . Because of the budget constraint, there must exist  $j_2 \in B$  such that  $x'_{i,j_2} > x^*_{i,j_2} \geq 0$ . In the following we focus only on the local allocation to  $j_1$  and  $j_2$ .

First observe that one unit of  $j_1$  is equivalent to  $p_{j_1}/p_{j_2}$  unit of  $j_2$  in price. By definition of optimality, it would not increase the utility if  $x'_{i,j_2}$  is decreased with a corresponding increase of  $x'_{i,j_1}$ . Therefore, the marginal utility gain by purchasing  $j_2$  with one unit money must be no less than that of purchasing  $j_1$ :

$$\frac{p_{j_1}}{p_{j_2}} \cdot u_{i,j_2}^{(1)}(x'_{i,j_2}) \geq u_{i,j_1}^{(1)}(x'_{i,j_1}).$$

Otherwise, we may decrease  $x'_{i,j_2}$  (and increase  $x'_{i,j_1}$  accordingly) to get higher utility for buyer  $i$ .

Using the last inequality and the monotonicity of  $u_{i,j}^{(1)}$ , we obtain

$$\frac{p_{j_1}}{p_{j_2}} \cdot u_{i,j_2}^{(1)}(x^*_{i,j_2}) > \frac{p_{j_1}}{p_{j_2}} \cdot u_{i,j_2}^{(1)}(x'_{i,j_2}) \geq u_{i,j_1}^{(1)}(x'_{i,j_1}) > u_{i,j_1}^{(1)}(x^*_{i,j_1}).$$

The inequality between the first and the last term implies that an increase in  $x^*_{i,j_2}$  (and a corresponding decrease in  $x^*_{i,j_1}$ ) will gain buyer  $i$  a higher utility, which is a contradiction to the optimality of  $x^*(P)$ .  $\square$

### 3 Dual Relation Between Buyers and Sellers

For the sake of presentation, we associate a seller with each goods, following [4]. We introduce a bipartite graph to describe the transactions between buyers and sellers.

**Definition 1.** *Give a price vector  $P$ , the transaction graph  $G(P) = (A, B, E)$  consists of vertices  $A, B$ , which represent the collection of buyers and sellers, respectively, and the edge set  $E$ . For any  $i \in A, j \in B, (i, j) \in E$  if and only if  $x^*_{i,j}(P) > 0$ .*

For  $i \in A$ , denote its neighborhood by  $\Gamma(i) = \{j \in B \mid (i, j) \in E\}$ . For  $j \in B$ , denote its neighborhood by  $\Phi(j) = \{i \in A \mid (i, j) \in E\}$ .

It follows immediately from Proposition 1 that, for any price vector, we can compute the incident edge set to  $i$ ,  $\Gamma(i)$ , as follows: First compute the optimal solution  $x'$  of (2). Then, drop the non-positive variables  $x'_{i,j}$  in the solution. For the remaining variables, repeat the above procedure until all solutions are positive. The remaining variables (with positive solutions) yield the corresponding neighborhood set  $\Gamma(i)$ . We summarize in the following proposition.

**Proposition 2.** *If  $x_{i,j} > 0$  in the last iteration, then  $x_{i,j}^* > 0$ .*

Note that in each iteration, at least one variable is dropped. Thus there can be at most  $m$  iterations. Hence, for any given price vector the above algorithm computes  $\Gamma(i)$  in  $O(m^3)$  time.

Now, for any given graph  $G$ , we define for each buyer  $i$  the “bias factor”  $\lambda_i$  as follows:

$$\lambda_i = \frac{e_i + \sum_{j \in \Gamma(i)} \epsilon_{i,j} p_j}{\sum_{j \in \Gamma(i)} a_{i,j}} \quad (3)$$

Note that in the last iteration of the algorithm to determine the transaction graph, we have  $\eta = 1/\lambda_i > 0$ , and for any  $j \in \Gamma(i)$ ,  $x_{i,j} = \frac{a_{i,j}}{\eta p_j} - \epsilon_{i,j} = \frac{a_{i,j}}{p_j} \lambda_i - \epsilon_{i,j} > 0$ .

At any time during the iteration, define the current value of  $1/\eta$  as the value of

$$\frac{e_i + \sum_{j \in \Gamma(i)} \epsilon_{i,j} p_j}{\sum_{j \in \Gamma(i)} a_{i,j}},$$

where  $\Gamma(i)$  is the set of remaining vertices in  $B$  at that time. We argue that the value of  $\eta$  can only increase in the iterative process. Note that, after each iteration except the last one, some variables  $x'_{i,j}$  become non-positive and thus  $j$  is dropped from  $\Gamma(i)$ . That means the new value of  $\eta$  must have become larger to decrease  $\frac{a_{i,j}}{\eta p_j} - \epsilon_{i,j}$  from being positive to zero or negative.

For any  $j \notin \Gamma(i)$ , when  $j$  is dropped from  $\Gamma(i)$  during an iteration,  $\frac{a_{i,j}}{\eta p_j} - \epsilon_{i,j} \leq 0$  for the value of  $\eta$  at that time. Since the value of  $\eta$  can only increase (reaching  $1/\lambda_i$  at the end), we conclude that  $\frac{a_{i,j}}{p_j} \lambda_i - \epsilon_{i,j} \leq 0$ . The above discussions lead to the following characterization of the transaction graph.

**Lemma 1.** *Given a price vector  $P > 0$ , the transaction graph  $G(P)$  satisfies the following conditions:*

(a)  $\forall j \in \Gamma(i)$ ,  $\lambda_i > \frac{\epsilon_{i,j}}{a_{i,j}} p_j$ , and  $\forall j \notin \Gamma(i)$ ,  $\lambda_i \leq \frac{\epsilon_{i,j}}{a_{i,j}} p_j$ . i.e.  $\Gamma(i) = \{j \in B \mid \lambda_i > \frac{\epsilon_{i,j}}{a_{i,j}} p_j\}$ .

(b)  $\forall j \in \Gamma(i)$ ,  $x_{i,j}^*(P) = \frac{a_{i,j}}{p_j} \lambda_i - \epsilon_{i,j}$ .

Moreover, of all the subsets of  $A$ ,  $\Gamma(i)$  is the only subset that can satisfy Equation (3) and Condition (a).

*Proof.* Conditions (a) and (b) follow from the above discussion. We only need to prove the uniqueness of  $\Gamma(i)$ . Assume without loss of generality that  $\frac{\epsilon_{i,1}}{a_{i,1}}p_1 \leq \dots \leq \frac{\epsilon_{i,m}}{a_{i,m}}p_m$ . Suppose there is another subset  $\Gamma'(i) \subseteq A$  satisfying Equation (3) and Condition (a). Assume  $\Gamma(i) = \{1, \dots, j\}$  and w.l.o.g. assume  $\Gamma'(i) = \{1, \dots, j, \dots, j+k\}$ ,  $k > 0$  (Discussion for the case  $k < 0$  is symmetric.). Let

$$\lambda_i = \frac{e_i + \sum_{j \in \Gamma(i)} \epsilon_{i,j} p_j}{\sum_{j \in \Gamma(i)} a_{i,j}}, \quad \lambda'_i = \frac{e_i + \sum_{j \in \Gamma'(i)} \epsilon_{i,j} p_j}{\sum_{j \in \Gamma'(i)} a_{i,j}}.$$

Thus,

$$\frac{\epsilon_{i,j}}{a_{i,j}} p_j < \lambda_i \leq \frac{\epsilon_{i,j+1}}{a_{i,j+1}} p_{j+1}, \quad \frac{\epsilon_{i,j+k}}{a_{i,j+k}} p_{j+k} < \lambda'_i \leq \frac{\epsilon_{i,j+k+1}}{a_{i,j+k+1}} p_{j+k+1}.$$

Note that  $\lambda_i \leq \frac{\epsilon_{i,j+1}}{a_{i,j+1}} p_{j+1}$  indicates that

$$\frac{e_i + \sum_{j \in \Gamma(i) \cup \{j+1\}} \epsilon_{i,j} p_j}{\sum_{j \in \Gamma(i) \cup \{j+1\}} a_{i,j}} \leq \frac{\epsilon_{i,j+1}}{a_{i,j+1}} p_{j+1}$$

Since  $\frac{\epsilon_{i,j+1}}{a_{i,j+1}} p_{j+1} \leq \frac{\epsilon_{i,j+2}}{a_{i,j+2}} p_{j+2} \leq \dots \leq \frac{\epsilon_{i,j+k}}{a_{i,j+k}} p_{j+k}$ , we have

$$\frac{e_i + \sum_{j \in \Gamma(i) \cup \{j+1, \dots, j+k\}} \epsilon_{i,j} p_j}{\sum_{j \in \Gamma(i) \cup \{j+1, \dots, j+k\}} a_{i,j}} \leq \frac{\epsilon_{i,j+k}}{a_{i,j+k}} p_{j+k}.$$

The left hand side of the above inequality is  $\lambda'_i$ , a contradiction to  $\frac{\epsilon_{i,j+k}}{a_{i,j+k}} p_{j+k} < \lambda'_i$ .  $\square$

Lemma 1 leads to an improved algorithm to compute  $\lambda_i$  and the transaction graph  $G(P)$  for any given price vector  $P$ .

**Lemma 2.** *For any given price vector  $P > 0$  and buyer  $i$ ,  $\lambda_i$  and  $\Gamma(i)$  can be computed in  $O(m \log m)$  time. Therefore, we can compute  $G(P)$  in time  $O(nm \log(m))$ .*

*Proof.* By Lemma 1, we know that  $\forall j \in \Gamma(i)$ ,  $\lambda_i > \frac{\epsilon_{i,j}}{a_{i,j}} p_j$ , and  $\forall j \notin \Gamma(i)$ ,  $\lambda_i \leq \frac{\epsilon_{i,j}}{a_{i,j}} p_j$ . Thus we design an algorithm as follows.

First, we calculate all  $\frac{\epsilon_{i,j}}{a_{i,j}} p_j$ ,  $j = 1, \dots, m$ , and sort them into non-decreasing order. Thus, the set  $\Gamma(i)$  has at most  $m$  different choices. For each candidate set of  $\Gamma(i)$ , calculate the current “bias factor”  $\lambda_i$  according to Equation (3), and check whether  $\Gamma(i)$  is equal to the set  $\{j \in B \mid \lambda_i > \frac{\epsilon_{i,j}}{a_{i,j}} p_j\}$ . The computation of  $\lambda_i$  can be done in  $O(1)$  steps per candidate set, and thus the dominating computation time is in the sorting of  $\frac{\epsilon_{i,j}}{a_{i,j}} p_j$ ,  $j = 1, 2, \dots, m$ , and the Lemma follows.  $\square$

From now on in this section, we restrict our discussion to the equilibrium price vector.

**Lemma 3.** *For the equilibrium price vector  $P$ , we have*

$$p_j = \frac{\sum_{i \in \Phi(j)} a_{i,j} \lambda_i}{q_j + \sum_{i \in \Phi(j)} \epsilon_{i,j}} \quad (4)$$

and  $\Phi(j) = \{i \in A \mid \frac{a_{i,j}}{\epsilon_{i,j}} \lambda_i > p_j\}$ .

*Proof.* By Lemma 1, we have  $(i, j) \in E \Leftrightarrow a_{i,j} \lambda_i > \epsilon_{i,j} p_j$ , thus  $\Phi(j) = \{i \in A \mid \frac{a_{i,j}}{\epsilon_{i,j}} \lambda_i > p_j\}$ . And also from Lemma 1,  $\forall (i, j) \in E$ ,  $x_{i,j} = \frac{a_{i,j}}{p_j} \lambda_i - \epsilon_{i,j}$ . According to the market clearance condition,  $q_j = \sum_{i \in \Phi(j)} x_{i,j}$ . Thus  $q_j = \sum_{i \in \Phi(j)} (\frac{a_{i,j}}{p_j} \lambda_i - \epsilon_{i,j}) \Rightarrow p_j = \frac{\sum_{i \in \Phi(j)} a_{i,j} \lambda_i}{q_j + \sum_{i \in \Phi(j)} \epsilon_{i,j}}$ .  $\square$

Lemma 3 presents a description of the optimal behavior of sellers. It allows use to obtain an algorithm to calculate  $p_j$  and  $\Phi(j)$  on the basis of  $\Lambda$ , where  $\Lambda = (\lambda_1, \dots, \lambda_n)$ .

**Lemma 4.** *Assume we know the vector  $\Lambda$ , then there exists the unique solution  $p_j$  and  $\Phi(j)$  for seller  $j$ , and it can be computed in  $O(n \log n)$  time.*

*Proof.* By Lemma 3,  $\Phi(j) = \{i \in A \mid \frac{a_{i,j}}{\epsilon_{i,j}} \lambda_i > p_j\}$ . We first sort all  $\frac{a_{i,j}}{\epsilon_{i,j}} \lambda_i$ ,  $i = 1, \dots, n$ , in a non-increasing order. Then  $\Phi(j)$  has at most  $n$  choices. For each candidate of  $\Phi(j)$ , calculate  $p_j$  according to Equation (4), where the sums can be obtained by preprocessing the prefix sum series in  $O(m)$  time.  $\Phi(j)$  is then determined by whether it is equal to  $\{i \in A \mid \frac{a_{i,j}}{\epsilon_{i,j}} \lambda_i > p_j\}$ .  $\square$

A dual relation is now established between the price vector  $P$  and the “bias factor”  $\Lambda$ :

$$\begin{aligned} P &\leftrightarrow \Lambda \\ p_j &= \frac{\sum_{i \in \Phi(j)} a_{i,j} \lambda_i}{q_j + \sum_{i \in \Phi(j)} \epsilon_{i,j}} \leftrightarrow \lambda_i = \frac{e_i + \sum_{j \in \Gamma(i)} \epsilon_{i,j} p_j}{\sum_{j \in \Gamma(i)} a_{i,j}} \\ \Phi(j) &= \{i \in A \mid \frac{a_{i,j}}{\epsilon_{i,j}} \lambda_i > p_j\} \leftrightarrow \Gamma(i) = \{j \in B \mid \lambda_i > \frac{\epsilon_{i,j}}{a_{i,j}} p_j\} \end{aligned}$$

Although the “bias factor” was first defined by the price vector (Equation (3)), we can treat it as a natural parameter for buyers as a dual to the price vector  $P$  for sellers: Each seller  $j$  decides which collection of buyers to sell his goods, according to the quantity  $q_j$  and the buyers’ “bias factor”  $\Lambda$  (dually, each buyer  $i$  decides which collection of goods to buy, according to his total endowment  $e_i$  and the sellers’ price  $P$ ). This property will be used in the next section to design an algorithm for finding the equilibrium solution.



## 4 Solution for Bounded Number of Buyers or Sellers

Our solution depends on the following lemma.

**Lemma 5.** *Given a bipartite graph  $G(A, B)$ , if for all  $i \in A, j \in B, \Gamma(i) \neq \emptyset, \Phi(j) \neq \emptyset$ , then the linear equation system*

$$\begin{cases} \lambda_i = \frac{e_i + \sum_{j \in \Gamma(i)} \epsilon_{i,j} p_j}{\sum_{j \in \Gamma(i)} a_{i,j}}, & i = 1, \dots, n \\ p_j = \frac{\sum_{i \in \Phi(j)} a_{i,j} \lambda_i}{q_j + \sum_{i \in \Phi(j)} \epsilon_{i,j}}, & j = 1, \dots, m \end{cases} \quad (5)$$

is non-degenerate.

*Proof.* From the equations, we have

$$\begin{aligned} \lambda_i &= \frac{e_i + \sum_{j \in \Gamma(i)} \epsilon_{i,j} p_j}{\sum_{j \in \Gamma(i)} a_{i,j}} \\ &= \frac{e_i}{\sum_{l \in \Gamma(i)} a_{i,l}} + \frac{1}{\sum_{l \in \Gamma(i)} a_{i,l}} \sum_{j \in \Gamma(i)} \epsilon_{i,j} \frac{\sum_{k \in \Phi(j)} a_{k,j} \lambda_k}{q_j + \sum_{k \in \Phi(j)} \epsilon_{k,j}} \\ &= \frac{e_i}{\sum_{l \in \Gamma(i)} a_{i,l}} + \frac{1}{\sum_{l \in \Gamma(i)} a_{i,l}} \sum_{j \in \Gamma(i)} \left( \sum_{k \in \Phi(j)} \frac{a_{k,j} \epsilon_{i,j} \lambda_k}{q_j + \sum_{l \in \Phi(j)} \epsilon_{l,j}} \right) \end{aligned}$$

Now we change the order of dummy  $j, k$  is the right side of the equation:

$$\begin{aligned} \lambda_i &= \frac{e_i}{\sum_{l \in \Gamma(i)} a_{i,l}} + \frac{1}{\sum_{l \in \Gamma(i)} a_{i,l}} \sum_{k: \Gamma(i) \cap \Gamma(k) \neq \emptyset} \left( \sum_{j: j \in \Gamma(i) \cap \Gamma(k)} \frac{a_{k,j} \epsilon_{i,j} \lambda_k}{q_j + \sum_{l \in \Phi(j)} \epsilon_{l,j}} \right) \\ &= \frac{e_i}{\sum_{l \in \Gamma(i)} a_{i,l}} + \frac{1}{\sum_{l \in \Gamma(i)} a_{i,l}} \sum_{k: \Gamma(i) \cap \Gamma(k) \neq \emptyset} \lambda_k \left( \sum_{j: j \in \Gamma(i) \cap \Gamma(k)} \frac{a_{k,j} \epsilon_{i,j}}{q_j + \sum_{l \in \Phi(j)} \epsilon_{l,j}} \right) \end{aligned}$$

i.e.

$$\left( \sum_{j \in \Gamma(i)} a_{i,j} \right) \lambda_i = e_i + \sum_{k: \Gamma(i) \cap \Gamma(k) \neq \emptyset} \lambda_k \left( \sum_{j: j \in \Gamma(i) \cap \Gamma(k)} \frac{a_{k,j} \epsilon_{i,j}}{q_j + \sum_{l \in \Phi(j)} \epsilon_{l,j}} \right)$$

Move  $\lambda_k$  to the left, we get an equation system of  $\lambda$ :  $F\lambda = e$ , here  $F = [f_{i,k}]_{m \times m}$  is a  $m \times m$  matrix,  $e = [e_1, \dots, e_m]^T$ , where

$$\begin{cases} f_{i,i} = \left( \sum_{j \in \Gamma(i)} a_{i,j} \right) - \sum_{j: j \in \Gamma(i)} \frac{a_{i,j} \epsilon_{i,j}}{q_j + \sum_{l \in \Phi(j)} \epsilon_{l,j}}, \\ f_{i,k} = - \sum_{j: j \in \Gamma(i) \cap \Gamma(k)} \frac{a_{k,j} \epsilon_{i,j}}{q_j + \sum_{l \in \Phi(j)} \epsilon_{l,j}}, & \text{if } \Gamma(i) \cap \Gamma(k) \neq \emptyset, k \neq i \\ f_{i,k} = 0, & \text{if } \Gamma(i) \cap \Gamma(k) = \emptyset \end{cases}$$

Now we show that  $\forall k, \sum_{i:i \neq k} |f_{i,k}| < f_{k,k}$ : In fact,

$$\begin{aligned}
& f_{k,k} - \sum_{i:i \neq k} |f_{i,k}| \\
&= \left( \sum_{j \in \Gamma(k)} a_{k,j} \right) - \sum_{i: \Gamma(i) \cap \Gamma(k) \neq \emptyset} \left( \sum_{j: j \in \Gamma(i) \cap \Gamma(k)} \frac{a_{k,j} \epsilon_{i,j}}{q_j + \sum_{l \in \Phi(j)} \epsilon_{l,j}} \right) \\
&= \left( \sum_{j \in \Gamma(k)} a_{k,j} \right) - \sum_{j: j \in \Gamma(k)} \left( \sum_{i: i \in \Phi(j)} \frac{a_{k,j} \epsilon_{i,j}}{q_j + \sum_{l \in \Phi(j)} \epsilon_{l,j}} \right) \\
&= \left( \sum_{j \in \Gamma(k)} a_{k,j} \right) - \sum_{j: j \in \Gamma(k)} \left( \frac{a_{k,j}}{q_j + \sum_{l \in \Phi(j)} \epsilon_{l,j}} \sum_{i: i \in \Phi(j)} \epsilon_{i,j} \right) \\
&= \sum_{j \in \Gamma(k)} a_{k,j} \left( 1 - \frac{\sum_{i: i \in \Phi(j)} \epsilon_{i,j}}{q_j + \sum_{l \in \Phi(j)} \epsilon_{l,j}} \right) \\
&= \sum_{j \in \Gamma(k)} \frac{a_{k,j} b_j}{q_j + \sum_{l \in \Phi(j)} \epsilon_{l,j}} > 0
\end{aligned}$$

So  $\text{rank}(F) = m$ , the linear equation system is non-degenerate.  $\square$

**Proposition 3.** *Given the transaction graph  $G$ ,  $P$  and  $\Lambda$  can be computed in  $O((m+n)^3)$  time.*

*Proof.* We establish  $(m+n)$  linear equations of  $\Lambda$  and  $P$  from graph  $G$ :

$$\begin{cases} \lambda_i = \frac{e_i + \sum_{j \in \Gamma(i)} \epsilon_{i,j} p_j}{\sum_{j \in \Gamma(i)} a_{i,j}}, & i = 1, \dots, n \\ p_j = \frac{\sum_{i \in \Phi(j)} a_{i,j} \lambda_i}{q_j + \sum_{i \in \Phi(j)} \epsilon_{i,j}}, & j = 1, \dots, m \end{cases}$$

By Lemma 5, the set of equations is non-degenerate for  $e, q$  corresponding to the equilibrium solution. It can be computed in  $O((m+n)^3)$  time.  $\square$

The number of bipartite graphs with  $m$  and  $n$  vertices on each side has in general  $2^{mn}$  different choices, which is exponential for both  $m$  and  $n$ . In the following, we show how to reduce this number to polynomial, when either  $m$  or  $n$  is bounded by a constant.

**Lemma 6.** *There is an algorithm solving the Market equilibrium problem with logarithmic utility functions in  $O((m + 2^m mn)^{2m+1} (m+n)^3)$  time.*

*Proof.* Due to Proposition 3, we only need to prove there are at most  $O((m + 2^m mn)^m)$  different possibilities of graph  $G$ , or equivalently, different choices of  $(\Gamma(1), \dots, \Gamma(n))$ , and that it takes  $O((m + 2^m mn)^{2m+1})$  time to generate all these graphs.

For each  $i \in A$  and subset  $\Delta \subseteq B$ , let  $\mathcal{I}_{i,\Delta}$  denote the system of  $m$  inequalities

$$\begin{cases} \frac{e_i + \sum_{k \in \Delta} \epsilon_{i,k} p_k}{\sum_{k \in \Delta} a_{i,k}} > \frac{\epsilon_{i,j}}{a_{i,j}} p_j, \forall j \in \Delta \\ \frac{e_i + \sum_{k \in \Delta} \epsilon_{i,k} p_k}{\sum_{k \in \Delta} a_{i,k}} \leq \frac{\epsilon_{i,j}}{a_{i,j}} p_j, \forall j \notin \Delta \end{cases} \quad (6)$$

If a  $(\Gamma(1), \Gamma(2), \dots, \Gamma(n))$  (where  $\Gamma(i) \subseteq B$ ) is a possible candidate for the transaction graph, then by Lemma 3.1, there must exist some  $P \in \mathbb{R}_+^m$  such that  $\mathcal{I}_{i,\Gamma(i)}$  holds for every  $i \in A$ .

For each  $i \in A$ , subset  $\Delta \subseteq B$  and  $j \in \Delta$ , let  $f_{i,\Delta,j}(p_1, p_2, \dots, p_m)$  denote the following linear function in variables  $p_1, \dots, p_m$ :

$$\frac{e_i + \sum_{k \in \Delta} \epsilon_{i,k} p_k}{\sum_{k \in \Delta} a_{i,k}} - \frac{\epsilon_{i,j}}{a_{i,j}} p_j.$$

Let  $N$  be the number of all triplets  $(i, \Delta, j)$ , then  $N \leq nm2^m$ . Let  $P \in \mathbb{R}_+^m$  be an unknown price vector, but that we know the answer  $\xi(P) \in \{>, \leq\}^N$  to all the binary comparison queries “ $f_{i,\Delta,j}(P) : 0$ ”. Then for each  $i$  we can easily determine all the subsets  $\Delta \subseteq B$  such that the system of inequalities  $\mathcal{I}_{i,\Delta}$  holds. By Lemma 3.1, there is in fact exactly one such  $\Delta$ , which we call  $\Gamma(i)$ . In this fashion, each possible  $\xi(P)$  leads to one candidate graph  $G(P)$ , as specified by  $(\Gamma(1), \dots, \Gamma(n))$ ; this computation can be done in time  $O(N)$ .

To prove the Lemma, it suffices to show that there are at most  $O(N^m)$  distinct  $\xi(P)$ ’s, and that they can be generated in time  $O(N^{2m+1})$ . In  $\mathbb{R}^m$ , consider the  $N + m$  hyperplanes  $f_{i,\Delta,j}(p_1, p_2, \dots, p_m) = 0$  and  $p_i = 0$ . They divide the space into regions depending on answers to the queries “ $f_{i,\Delta,j}(p_1, p_2, \dots, p_m) > 0$ ?” and “ $p_i > 0$ ?”. For any region in  $\mathbb{R}_+^m$ , all the points  $P$  in it share the same  $\xi(P)$ . It is well known that there are at most  $O((N + m)^m)$  regions. We can easily enumerate all these regions and compute their  $\xi(P)$ ’s in time  $O((N + m)^m(N + m)^{m+1})$ , by adding hyperplanes one at a time and updating the regions using linear programming in  $\mathbb{R}^m$ .  $\square$

Lemma 6 solves the Market equilibrium problem in the price space  $\mathbb{R}^m$ . By the dual relationship, we can also solve the problem in the “bias factor” space  $\mathbb{R}^n$  in an almost verbatim manner.

**Lemma 7.** *There is an algorithm solving the Market equilibrium problem with logarithmic utility functions in  $O((m + 2^n mn)^{2m+1}(m + n)^3)$  time.*

From Lemma 6 and Lemma 7, we have the following conclusion.

**Theorem 1.** *If the number of buyers, or goods, is bounded by a constant, then there is a polynomial time algorithm solving the Market equilibrium problem with logarithmic utility functions.*

## 5 Conclusion and Discussion

In this paper, we have derived a duality relation for the market equilibrium problem with logarithmic utility functions, and use it to develop a polynomial time algorithm when the number of either the buyers or the sellers is bounded by a constant. The techniques developed may be of some help for general concave utility functions.

We note that, recently, Devanur and Vazirani made some extension to design a PTAS for another special type of value functions for the buyers as they considered the concave utility function very difficult to deal with by their approach [6]. At the same time, Codenotti and Varadarajan [2] studied how to compute equilibrium price for Leontief utility function.

## References

1. K. K. Arrow, G. Debreu, *Existence of An Equilibrium for a Competitive Economy*, *Econometrica*, V. 22, 265-290, 1954.
2. B. Codenotti, K. Varadarajan, *Efficient Computation of Equilibrium Prices for Market with Leontief Utilities*, To appear in ICALP 2004.
3. Xiaotie Deng, C. H. Papadimitriou, S. Safra, *On the Complexity of Equilibria*, STOC 2002, 67-71. Journal version: Xiaotie Deng, C. H. Papadimitriou, S. Safra, *On the Complexity of Price Equilibrium*, *Journal of Computer and System Sciences*, V. 67(2), 311-324, 2003.
4. N. Devanur, C. H. Papadimitriou, A. Saberi, V. V. Vazirani, *Market Equilibrium via a Primal-Dual-Type Algorithm*, FOCS 2002, 389-395.
5. N. Devanur, V. Vazirani, *An Improved Approximation Scheme for Computing Arrow-Debreu Prices for the Linear Case*, FSTTCS 2003, 149-155.
6. N. Devanur, V. Vazirani, *The Spending Constraint Model for Market Equilibrium: Algorithmic, Existence and Uniqueness Results*, STOC 2004.
7. D. Gale, *The Theory of Linear Economic Models*, McGraw Hill, N.Y., 1960.
8. K. Jain, M. Mahdian, A. Saberi, *Approximating Market Equilibria*, APPROX 2003, pp.98-108.
9. S. Kapoor, R. Garg, *Auction Algorithms for Market Equilibrium*, STOC 2004.
10. B.T. Polyak: "Introduction to Optimization", Opt. Software Inc., 1987.
11. H. E. Scarf, *The Computation of Economic Equilibria* (with collaboration of T. Hansen), Cowles Foundation Monograph No. 24, Yale University Press, 1973.

# Hardness and Approximation Results for Packing Steiner Trees

Joseph Cheriyan<sup>1\*</sup> and Mohammad R. Salavatipour<sup>2\*\*</sup>

<sup>1</sup> Department of Combinatorics and Optimization, University of Waterloo,  
Waterloo, Ontario N2L3G1, Canada  
[jcheriyan@math.uwaterloo.ca](mailto:jcheriyan@math.uwaterloo.ca)

<sup>2</sup> Department of Computing Science, University of Alberta,  
Edmonton, Alberta T6G2H1, Canada  
[mreza@cs.ualberta.ca](mailto:mreza@cs.ualberta.ca)

**Abstract.** We study approximation algorithms and hardness of approximation for several versions of the problem of packing Steiner trees. For packing edge-disjoint Steiner trees of undirected graphs, we show APX-hardness for 4 terminals. For packing Steiner-node-disjoint Steiner trees of undirected graphs, we show a logarithmic hardness result, and give an approximation guarantee of  $O(\sqrt{n} \log n)$ , where  $n$  denotes the number of nodes. For the directed setting (packing edge-disjoint Steiner trees of directed graphs), we show a hardness result of  $\Omega(m^{\frac{1}{3}-\epsilon})$  and give an approximation guarantee of  $O(m^{\frac{1}{2}+\epsilon})$ , where  $m$  denotes the number of edges. The paper has several other results.

## 1 Introduction

We study approximation algorithms and hardness (of approximation) for several versions of the problem of packing Steiner trees. Given an undirected graph  $G = (V, E)$  and a set of *terminal* nodes  $T \subseteq V$ , a *Steiner tree* is a connected, acyclic subgraph that contains all the terminal nodes (nonterminal nodes, which are called *Steiner nodes*, are optional). The basic problem of Packing Edge-disjoint Undirected Steiner trees (**PEU** for short) is to find as many edge-disjoint Steiner trees as possible. Besides PEU, we study some other versions (see below for details).

The PEU problem in its full generality has applications in VLSI circuit design (e.g., see [11,21]). Other applications include multicasting in wireless networks (see [7]) and broadcasting large data streams (such as videos) over the Internet (see [14]). There is significant motivation from the areas of graph theory and combinatorial optimization. Menger's theorem on packing edge-disjoint  $s, t$ -paths [5] corresponds to the special case of packing edge-disjoint Steiner trees on two

---

\* Supported by NSERC grant No. OGP0138432.

\*\* Research done while a postdoc at the Dept. of Combinatorics and Optimization, University of Waterloo. Supported by an NSERC postdoctoral fellowship and the Dept. of Comb. and Opt., University of Waterloo.

terminal nodes (i.e.,  $T = \{s, t\}$ ). Another special case is when all the nodes are terminals (i.e.,  $T = V$ ). Then the problem is to find a maximum set of edge-disjoint spanning trees. This topic was studied in the 1960's by graph theorists, and a min-max theorem was developed by Tutte and independently by Nash-Williams [5]. Subsequently, Edmonds and Nash-Williams derived such results in the more general setting of the matroid intersection theorem. One consequence is that efficient algorithms are available via the matroid intersection algorithm for the case of  $T = V$ . (Note that most problems on packing Steiner trees are NP-hard, so results from matroid optimization do not apply directly.) A set of nodes  $S$  is said to be  $\lambda$ -edge connected if there exist  $\lambda$  edge-disjoint paths between every two nodes of  $S$ . An easy corollary of the min-max theorem is that if the node set  $V$  is  $2k$ -edge connected, then the graph has  $k$  edge-disjoint spanning trees. Recently, Kriesell [19] conjectured an exciting generalization: If the set of terminals is  $2k$ -edge connected, then there exist  $k$  edge-disjoint Steiner trees. He proved this for Eulerian graphs (by an easy application of the splitting-off theorem). Note that a constructive proof of this conjecture may give a 2-approximation algorithm for PEU. Jain, Mahdian, and Salavatipour [14] gave an approximation algorithm with guarantee (roughly)  $\frac{|T|}{4}$ . Moreover, using a versatile and powerful proof technique (that we will borrow and apply in the design of our algorithms), they showed that the fractional version of PEU has an  $\alpha$ -approximation algorithm if and only if the minimum-weight Steiner tree problem has an  $\alpha$ -approximation algorithm (Theorem 4.1 in [14]). The latter problem is well studied and is known to be APX-hard. It follows that PEU is APX-hard (Corollary 4.3 in [14]). Kaski [16] showed that the problem of finding two edge-disjoint Steiner trees is NP-hard, and moreover, problem PEU is NP-hard even if the number of terminals is 7. Frank et al. [8] gave a 3-approximation algorithm for a special case of PEU (where no two Steiner nodes are adjacent). Very recently Lau [20], based on the result of Frank et al. [8], has given an  $O(1)$  approximation algorithm for PEU (but Kriesell's conjecture remains open).

Using the fact that PEU is APX-hard, Floréen et al. [7] showed that packing Steiner-node-disjoint Steiner trees (see problem PVU defined below) is APX-hard. They raised the question whether this problem is in the class APX. Also, they showed that the special case of the problem with 4 terminal nodes is NP-hard.

## Our Results:

We use  $n$  and  $m$  to denote the number of nodes and edges, respectively. The underlying assumption for most of our hardness results is  $P \neq NP$ .

- **Packing Edge Disjoint Undirected Steiner Trees (PEU):** For this setting, we show that the maximization problem with only four terminal nodes is APX-hard. This result appears in Section 3. (An early draft of our paper also proved, independently of [16], that finding two edge-disjoint Steiner trees is NP-hard.)
- **Packing Vertex Capacitated Undirected Steiner Trees (PVCU):** We are given an undirected graph  $G$ , a set  $T \subseteq V$  of terminals, and a positive vertex capacity  $c_v$  for each Steiner vertex  $v$ . The goal is to find the maximum number of Steiner

trees such that the total number of trees containing each Steiner vertex  $v$  is at most  $c_v$ . The special case where all the capacities are 1 is the problem of *Packing Vertex Disjoint Undirected Steiner Trees* (**PVU** for short). Note that here and elsewhere we use “vertex disjoint Steiner trees” to mean trees that are disjoint on the *Steiner nodes* (and of course they contain all the terminals). We show essentially the same hardness results for PVU as for PEU, that is, finding two vertex disjoint Steiner trees is NP-hard and the maximization problem for a constant number of terminals is APX-hard. For an arbitrary number of terminals, we prove an  $\Omega(\log n)$ -hardness result (lower bound) for PVU, and give an approximation guarantee (upper bound) of  $O(\sqrt{n} \log n)$  for PVCU, by an LP-based rounding algorithm. This shows that PVU is significantly harder than PEU, and this settles (in the negative) an open question of Floréen et al. [7] (mentioned above). These results appear in Section 3. Although the gap for PVU between our hardness result and the approximation guarantee is large, we tighten the gap for another natural generalization of PVU, namely *Packing Vertex Capacitated Priority Steiner Trees* (PVCU-priority for short), which is motivated by the Quality of Service in network design problems (see [4] for applications). For this priority version, we show a lower-bound of  $\Omega(n^{\frac{1}{3}-\epsilon})$  on the approximation guarantee; moreover, our approximation algorithm for PVCU extends to PVCU-priority to give a guarantee of  $O(n^{\frac{1}{2}+\epsilon})$ ; see Subsection 3.1. (Throughout, we use  $\epsilon$  to denote any positive real number.)

- *Packing Edge/Vertex Capacitated Directed Steiner Trees* (**PECD** and **PVCD**): Consider a directed graph  $G(V, E)$  with a positive capacity  $c_e$  for each edge  $e$ , a set  $T \subseteq V$  of terminals, and a specified root vertex  $r$ ,  $r \in T$ . A *directed Steiner tree* rooted at  $r$  is a subgraph of  $G$  that contains a directed path from  $r$  to  $t$ , for each terminal  $t \in T$ . In the problem of Packing Edge Capacitated Directed Steiner trees (**PECD**) the goal is to find the maximum number of directed Steiner trees rooted at  $r$  such that for every edge  $e \in E$  the total number of trees containing  $e$  is at most  $c_e$ . We prove an  $\Omega(m^{\frac{1}{3}-\epsilon})$ -hardness result even for unit-capacity PECD (i.e., packing edge-disjoint directed Steiner trees), and also provide an approximation algorithm with a guarantee of  $O(m^{\frac{1}{2}+\epsilon})$ . Moreover, we show the NP-hardness of the problem of finding two edge-disjoint directed Steiner trees with only three terminal nodes. We also consider the problem of Packing Vertex Capacitated Directed Steiner trees (**PVCD**), where instead of capacities on the edges we have a capacity  $c_v$  on every Steiner node  $v$ , and the goal is to find the maximum number of directed Steiner trees such that the number of trees containing any Steiner node  $v$  is at most  $c_v$ . For directed graphs, PECD and PVCD are quite similar and we get the following results on the approximation guarantee for the latter problem: a lower-bound of  $\Omega(n^{\frac{1}{3}-\epsilon})$  (even for the special case of unit capacities), and an upper bound of  $O(n^{\frac{1}{2}+\epsilon})$ . These results appear in Section 2.

In summary, with the exception of PVCU, the approximation guarantees (upper bounds) and hardness results (lower bounds) presented in this paper and the previous works [7,14,16,20] are within the same class with respect to the classification in Table 10.2 in [1].

## Comments:

Several of our proof techniques are inspired by results for disjoint-paths problems in the papers by Guruswami et al. [9], Baveja and Srinivasan [2], and Kolliopoulos and Stein [17]. (In these problems, we are given a graph and a set of source-sink pairs, and the goal is to find a maximum set of edge/node disjoint source-sink paths.) To the best of our knowledge, there is no direct relation between Steiner tree packing problems and disjoint-paths problems – neither problem is a special case of the other one. (In both problems, increasing the number of terminals seems to increase the difficulty for approximation, but each Steiner tree has to connect together all the terminals, whereas in the disjoint-paths problems the goal is to connect as many source-sink pairs as possible by disjoint paths.) There is a common generalization of both these problems, namely, the problem of packing Steiner trees with different terminal sets (given  $\ell$  sets of terminals  $T_1, T_2, \dots, T_\ell$ ,  $\ell$  polynomial in  $n$ , find a maximum set of edge-disjoint, or Steiner-node-disjoint, Steiner trees where each tree contains one of the terminal sets  $T_i$  ( $1 \leq i \leq \ell$ )). We chose to focus on our special cases (with identical terminal sets) because our hardness results are of independent interest, and moreover, the best approximation guarantees for the special cases may not extend to the general problems.

For the problems PVCU, PVCU-priority, PECD, and PVCD, we are not aware of any previous results on approximation algorithms or hardness results other than [7], although there is extensive literature on approximation algorithms for the corresponding *minimum-weight* Steiner tree problems (e.g., [3] for minimum-weight directed Steiner trees and [12] for minimum-node-weighted Steiner trees).

Due to space constraints, most of our proofs are deferred to the full version of the paper. Very recently, we have obtained new results that substantially decrease the gap between the upper bounds and the lower bounds for PVU; these results will appear elsewhere.

## 2 Packing Directed Steiner Trees

In this section, we study the problem of packing directed steiner trees. We show that Packing Vertex Capacitated Directed Steiner trees (PVCD) and the edge capacitated version (PECD) are as hard as each other in terms of approximability (similarly for the unit capacity cases). Then we present the hardness results for PECD with unit capacities (i.e., edge-disjoint directed case), which immediately implies similar hardness results for PVCD with unit capacities (i.e., vertex-disjoint directed case). We also present an approximation algorithm for PECD which implies a similar approximation algorithm for PVCD. The proof of the following theorem is easy. The idea for the first direction is to take the line graph, and for the second one is to split every vertex into two adjacent vertices.

**Theorem 1.** *Given an instance  $I = (G(V, E), T \subseteq V, k)$  of PECD (of PVCD), there is an instance  $I' = (G'(V', E'), T' \subseteq V, k)$  of PVCD (of PECD) with*



$|G'| = \text{poly}(|G|)$ , such that  $I$  has  $k$  directed Steiner trees satisfying the capacities of the edges (vertices) if and only if  $I'$  has  $k$  directed Steiner trees satisfying the capacities of the vertices (edges).

## 2.1 Hardness Results

First we prove that PVCD with unit capacities is NP-hard even in the simplest non-trivial case where there are only three terminals (one root  $r$  and two other terminals) and we are asked to find only 2 vertex-disjoint Steiner trees. The problem becomes trivially easy if any of these two conditions is tighter, i.e., if the number of terminals is reduced to 2 or the number of Steiner trees that we have to find is reduced to 1. If the number of terminals is arbitrary, then we show that PVCD with unit capacities is NP-hard to approximate within a guarantee of  $O(n^{\frac{1}{3}-\epsilon})$  for any  $\epsilon > 0$ . The proof is relatively simple and does *not* rely on the PCP theorem. Also, as we mentioned before, both of these hardness results carry over to PECD. For both reductions, we use the following well-known NP-hard problem (see [10]):

**PROBLEM: 2DIRPATH:**

**INSTANCE:** A directed graph  $G(V, E)$ , distinct vertices  $x_1, y_1, x_2, y_2 \in V$ .

**QUESTION:** Are there two vertex-disjoint directed paths, one from  $x_1$  to  $y_1$  and the other from  $x_2$  to  $y_2$  in  $G$ ?

**Theorem 2.** *Given an instance  $I$  of PVCD with unit capacities and only three terminals, it is NP-hard to decide if it has 2 vertex-disjoint directed Steiner trees.*

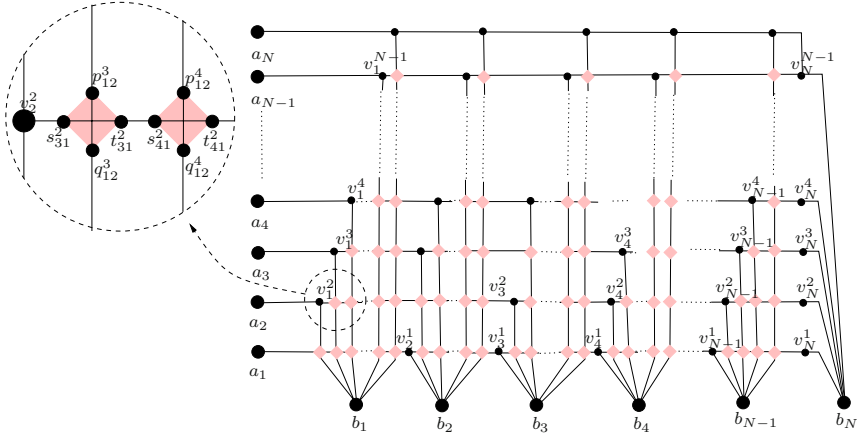
From Theorems 1 and 2, it follows that:

**Theorem 3.** *Given an instance  $I$  of PECD with unit capacities and only three terminals, it is NP-hard to decide if it has 2 edge-disjoint directed Steiner trees.*

Now we show that, unless  $P=NP$ , any approximation algorithm for PVCD with unit capacities has a guarantee of  $\Omega(n^{\frac{1}{3}-\epsilon})$ .

**Theorem 4.** *Given an instance of PVCD with unit capacities, it is NP-hard to approximate the solution within  $O(n^{\frac{1}{3}-\epsilon})$  for any  $\epsilon > 0$ .*

*Proof.* We use a reduction from the 2DIRPATH problem. Our proof is inspired by a reduction used in [9] for the edge-disjoint path problem. Assume that  $I = (G, x_1, y_1, x_2, y_2)$  is an instance of 2DIRPATH and let  $\epsilon > 0$  be given. We construct a directed graph  $H$ . First we construct a graph  $G'$  whose underlying structure is shown in Figure 1. For  $N = |V(G)|^{1/\epsilon}$ , create two sets of vertices  $A = \{a_1, \dots, a_N\}$  and  $B = \{b_1, \dots, b_N\}$ . In the figure, all the edges are directed from top to bottom and from left to right. For each grey box, there is a vertex at each of the four corners, and there are two edges, from left to right and from top to bottom. This graph may be viewed as the union of  $N$  vertex-disjoint directed trees  $T_1, \dots, T_N$ , where  $T_i$  is rooted at  $a_i$  and has paths to all the vertices in  $B - \{b_i\}$ . Each tree  $T_i$  consists of one horizontal path  $H^i$ , which is essentially the



**Fig. 1.** Construction of  $H$ : each grey box will be replaced with a copy of  $G$

$i$ th horizontal row above  $b_j$ 's, and starts with  $a_i, v_1^i, \dots$  and ends in  $v_N^i$ , together with  $N - 1$  vertical paths  $P_j^i$  ( $1 \leq j \neq i \leq N$ ), such that each of these vertical paths branches out from the horizontal path, starting at vertex  $v_j^i$  and ending at vertex  $b_j \in B$ . Each vertex  $v_j^i$  is in the  $i$ th horizontal row, and is the start vertex of a vertical path that ends at  $b_j$ ; note that there are no vertices  $v_i^i$ . Also, note that each grey box corresponds to a triple  $(i, j, \ell)$  where the box is in the  $i$ th horizontal line and is in the vertical path that starts at  $v_j^\ell$  and ends at  $b_j$ ; the corner vertices of the grey box are labeled  $s_{\ell j}^i, t_{\ell j}^i, p_{ji}^\ell, q_{ji}^\ell$  for the left, right, top, and bottom corners, respectively. More specifically, for  $T_1$  the horizontal and vertical paths are:  $H^1 = a_1, s_{21}^1, t_{21}^1, \dots, s_{N1}^1, t_{N1}^1, v_2^1, s_{32}^1, t_{32}^1, \dots, v_3^1, \dots, v_N^1$ , and  $P_j^1 = v_j^1, b_j$ , for  $2 \leq j \leq N$ . For  $T_2$  the horizontal and vertical paths are:  $H^2 = a_2, v_1^2, s_{31}^2, t_{31}^2, \dots, s_{N2}^2, t_{N2}^2, v_3^2, \dots, v_N^2$ , and  $P_j^2 = v_j^2, p_{j1}^2, q_{j1}^2, b_j$  (for  $1 \leq j \neq 2 < N$ ), and  $P_N^2 = v_N^2, b_N$ . In general, for  $T_i$ :

- $H^i = a_i, v_1^i, s_{(i+1)1}^i, t_{(i+1)1}^i, \dots, s_{N1}^i, t_{N1}^i, v_2^i, s_{(i+1)2}^i, t_{(i+1)2}^i, \dots, s_{N2}^i, t_{N2}^i, v_3^i, \dots, v_{N-1}^i, s_{(i+1)(N-1)}^i, t_{(i+1)(N-1)}^i, \dots, s_{N(N-1)}^i, t_{N(N-1)}^i, v_N^i$ ,
- For  $j \neq i < N$ :  $P_j^i = v_j^i, p_{j(i-1)}^i, q_{j(i-1)}^i, p_{j(i-2)}^i, q_{j(i-2)}^i, \dots, p_{j1}^i, q_{j1}^i, b_j$ , and  $P_N^i = v_N^i, b_N$ .

Graph  $H$  is obtained from  $G'$  by making the following modifications:

- For each grey box, with corresponding triple say  $(i, j, \ell)$  and with vertices  $s_{\ell j}^i, t_{\ell j}^i, p_{ji}^\ell, q_{ji}^\ell$ , we first remove the edges  $s_{\ell j}^i t_{\ell j}^i$  and  $p_{ji}^\ell q_{ji}^\ell$ , then we place a copy of graph  $G$  and identify vertices  $x_1, y_1, x_2$ , and  $y_2$  with  $s_{\ell j}^i, t_{\ell j}^i, p_{ji}^\ell$ , and  $q_{ji}^\ell$ , respectively.
- Add a new vertex  $r$  (root) and create directed edges from  $r$  to  $a_1, \dots, a_N$ .
- Create  $N$  new directed edges  $a_i b_i$ , for  $1 \leq i \leq N$ .

The set of terminals (for the directed Steiner trees) is  $B \cup \{r\} = \{b_1, \dots, b_N, r\}$ . The proof follows from the following two claims (we omit their proofs).

**Claim 1:** If  $I$  is a “Yes” instance of 2DIRPATH, then  $H$  has  $N$  vertex-disjoint directed Steiner trees.

**Claim 2:** If  $I$  is a “No” instance, then  $H$  has exactly 1 vertex-disjoint directed Steiner tree.

The number of copies of  $G$  in the construction of  $H$  is  $O(N^3)$  where  $N = |V(G)|^{1/\epsilon}$ . So the number of vertices in  $H$  is  $O(N^{3+\epsilon})$ . By Claims 1 and 2 it is NP-hard to decide if  $H$  has at least  $N$  or at most one directed Steiner trees. This creates a gap of  $\Omega(n^{\frac{1}{3}-\epsilon})$ .  $\square$

For PECD with unit capacities we use a reduction very similar to the one we presented for PVCD. The only differences are: (i) the instance that we use as the building block in our construction (corresponding to graph  $G$  above) is an instance of another well-known NP-hard problem, namely edge-disjoint 2DIRPATH (instead of vertex-disjoint), (ii) the parameter  $N$  above is  $|E(G)|^{1/\epsilon}$ . Using this reduction we can show:

**Theorem 5.** *Given an instance of PECD with unit capacities, it is NP-hard to approximate the solution within  $O(m^{\frac{1}{3}-\epsilon})$  for any  $\epsilon > 0$ .*

## 2.2 Approximation Algorithms

In this section we show that, although PECD is hard to approximate within a ratio of  $O(m^{1/3-\epsilon})$ , there is an approximation algorithm with a guarantee of  $O(m^{\frac{1}{2}+\epsilon})$  (details in Theorem 7). The algorithm is LP-based with a simple rounding scheme similar to those in [2,17]. The main idea of the algorithm is to start with one of the known approximation algorithms for finding a Minimum-weight Directed Steiner Tree. Using this and an extension of Theorem 4.1 in [14], we obtain an approximate solution to the fractional version of PECD. After that, a simple randomized rounding algorithm yields an integral solution. A similar method yields an approximation algorithm for PVCD that has a guarantee of  $O(n^{\frac{1}{2}+\epsilon})$ .

We may formulate PECD as an integer program (IP). In the following,  $\mathcal{F}$  denotes the collection of all directed Steiner trees in  $G$ .

$$\begin{aligned} & \text{maximize} && \sum_{F \in \mathcal{F}} x_F \\ & \text{subject to} && \forall e \in E : \sum_{F: e \in F} x_F \leq c_e \\ & && \forall F \in \mathcal{F} : x_F \in \{0, 1\} \end{aligned} \tag{1}$$

The *fractional packing edge capacitated directed Steiner tree* problem (fractional PECD, for short) is the linear program (LP) obtained by relaxing the integrality condition in the above IP to  $x_F \geq 0$ . For any instance  $I$  of the (integral) packing problem, we denote the fractional instance by  $I_f$ . The proof of Theorem 4.1 in [14] may be adapted to prove the following:

**Theorem 6.** *There is an  $\alpha$ -approximation algorithm for fractional PECD if and only if there is an  $\alpha$ -approximation algorithm for the minimum (edge weighted) directed Steiner tree problem.*

Charikar et al. [3] gave an  $O(n^\epsilon)$ -approximation algorithm for the minimum-weight directed Steiner tree problem. This, together with Theorem 6 implies:

**Corollary 1.** *There is an  $O(n^\epsilon)$ -approximation algorithm for fractional PECD.*

The key lemma in the design of our approximation algorithm for PECD is as follows.

**Lemma 1.** *Let  $I$  be an instance of PECD, and let  $\varphi^*$  be the (objective) value of a (not necessarily optimal) feasible solution  $\{x_F^* : F \in \mathcal{F}\}$  to  $I_f$  such that the number of non-zero  $x_F^*$ 's is polynomially bounded. Then, we can find in polynomial time, a solution to  $I$  with value at least  $\Omega(\max\{\varphi^*/\sqrt{m}, \min\{\varphi^{*2}/m, \varphi^*\})$ .*

**Theorem 7.** *Let  $I$  be an instance of PECD. Then for any  $\epsilon > 0$ , we can find in polynomial time a set of directed Steiner trees (satisfying the edge capacity constraints) of size at least  $\Omega(\max\{\varphi_f/m^{\frac{1+\epsilon}{2}}, \min\{\varphi_f^2/m^{1+\epsilon}, \varphi_f/m^{\epsilon/2}\})$ , where  $\varphi_f$  is the optimal value of the fractional instance  $I_f$ .*

*Proof.* Let  $I_f$  be the fractional instance. By Corollary 1, we can find an approximate solution  $\varphi^*$  for  $I_f$  such that  $\varphi^* \geq c\varphi_f/m^{\epsilon/2}$  for some constant  $c$  and the given  $\epsilon > 0$ . Furthermore, the approximate solution contains only a polynomial number of Steiner trees with non-zero fractional values (this follows from the proof of Theorem 6 which is essentially the same as Theorem 4.1 in [14]). If we substitute  $\varphi^*$  in Lemma 1 we obtain an approximation algorithm that finds a set  $\mathcal{F}'$  of directed Steiner trees such that  $\mathcal{F}'$  has the required size.  $\square$

For PVCD we do the following. Given an instance  $I$  of PVCD with graph  $G(V, E)$  and  $T \subseteq V$  (with  $|V| = n$  and  $|E| = m$ ), we first use the reduction presented in Theorems 1 to produce an instance  $I'$  of PECD with graph  $G'(V', E')$  and  $T' \subseteq V'$ . By the construction of  $G'$  we have  $|V'| = 2|V| = 2n$  and there are only  $n$  edges in  $E'$  with bounded capacities (corresponding to the vertices of  $G$ ). Therefore, if we use the algorithm of Theorem 7, the number of bad events will be  $n$ , rather than  $m$ . Using this observation we have the following:

**Theorem 8.** *Let  $I$  be an instance of PVCD. Then for any  $\epsilon > 0$  we can find in polynomial time a set of directed Steiner trees (satisfying the vertex capacity constraints) of size at least  $\Omega(\max\{\varphi_f/n^{\frac{1+\epsilon}{2}}, \min\{\varphi_f^2/n^{1+\epsilon}, \varphi_f/n^{\epsilon/2}\})$ , where  $\varphi_f$  is the optimal value of the fractional instance  $I_f$ .*

### 3 Packing Undirected Steiner Trees

For packing edge-disjoint undirected Steiner trees (PEU), Jain et al. [14] showed that the (general) problem is APX-hard, and Kaski [16] showed the special case

of the problem with only 7 terminal nodes is NP-hard. Here we show that PEU is APX-hard even when there are only 4 terminals. In an early draft of this paper we also showed, independently of [16], that finding two edge-disjoint Steiner trees is NP-hard. Both of these hardness results carry over (using similar constructions) to PVU. The following observation will be used in our proofs:

**Observation 2.** *For any solution to any of our Steiner tree packing problems, we may assume that: (1) In any Steiner tree, none of the leaves is a Steiner node (otherwise we simply remove it). (2) Every Steiner node with degree 3 belongs to at most one Steiner tree.*

Using this observation, the proof of NP-hardness for finding two edge-disjoint Steiner trees implies the following theorem.

**Theorem 9.** *Finding 2 vertex-disjoint undirected Steiner trees is NP-hard.*

**Theorem 10.** *PEU is APX-hard even if there are only 4 terminals.*

*Proof.* We use a reduction from Bounded 3-Dimensional Matching (B3DM). Assume that we are given three disjoint sets  $X, Y, Z$  (each corresponding to one part of a 3-partite graph  $G$ ), with  $|X| = |Y| = |Z| = n$ , and a set  $E \subseteq X \times Y \times Z$  containing  $m$  triples. Furthermore, we assume that each vertex in  $X \cup Y \cup Z$  belongs to at most 5 triples. It is known [15] that there is an absolute constant  $\epsilon_0 > 0$  such that it is NP-hard to distinguish between instances of B3DM where there is a perfect matching (i.e.,  $n$  vertex-disjoint triples) and those in which every matching (set of vertex-disjoint triples) has size at most  $(1 - \epsilon_0)n$ . Assume that  $x_1, \dots, x_n, y_1, \dots, y_n$ , and  $z_1, \dots, z_n$  are the nodes of  $X, Y$ , and  $Z$ , respectively. We construct a graph  $H$  which consists of:

- 4 terminals  $t_x, t_y, t_z$ , and  $t_{yz}$ .
- Non-terminals  $x_1, \dots, x_n, y_1, \dots, y_n$ , and  $z_1, \dots, z_n$  (corresponding to the nodes in  $X, Y, Z$ ),  $x'_1, \dots, x'_{m-n}, y'_1, \dots, y'_{m-n}$ , and  $z'_1, \dots, z'_n$ .
- Two non-terminals  $U$  and  $W$ .
- Edges  $t_x x_i, t_y y_i, t_z z'_i, z'_i z_i$ , and  $t_{yz} z'_i$ , for  $1 \leq i \leq n$ .
- Edges  $t_x x'_i, x'_i y'_i, x'_i U, y'_i t_y$ , and  $y'_i t_{yz}$ , for  $1 \leq i \leq m - n$ .
- $m - n$  parallel edge from  $W$  to  $t_z$ .
- For each triple  $e_q = x_i y_j z_k \in E$ , 3 non-terminals  $v_c^q, v_x^q, v_z^q$  and the following edges:  $v_x^q x_i, v_z^q z_k, v_c^q v_x^q, v_c^q y_j, v_c^q v_z^q, v_x^q U$ , and  $v_z^q W$ .

See Figure 2. We can show that (a) [completeness] if  $G$  has a perfect matching then  $H$  has  $m$  edge-disjoint Steiner trees and (b) [soundness] if every matching in  $G$  has size at most  $(1 - \epsilon_0)n$  then  $H$  has at most  $(1 - \epsilon_1)m$  edge-disjoint Steiner trees, for  $\epsilon_1 \geq \epsilon_0/110$ .  $\square$

The constant 110 in the above theorem is not optimal. We can find explicit lower bounds for the hardness of PEU with constant number of terminals using the known hardness results for  $k$ DM, for higher values of  $k$ . For instance, Hazan et al. [13] proved that 4DM (with upper bounds on the degree of each vertex)

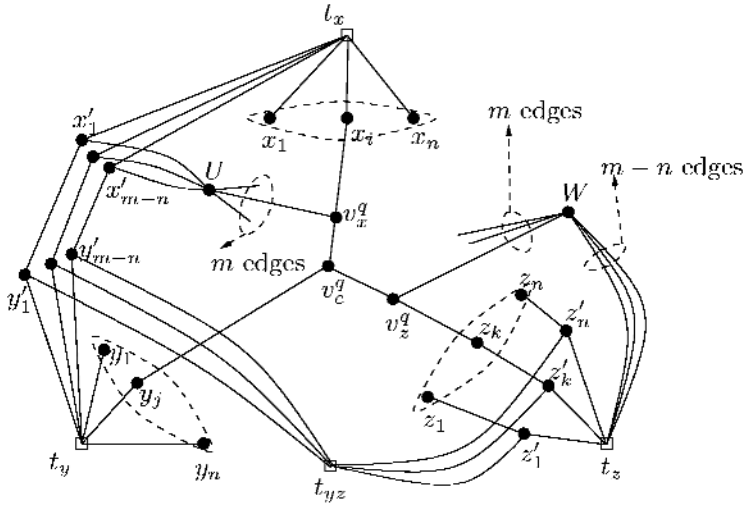


Fig. 2. Construction with 4 terminals from B3DM

is hard to approximate within a factor  $\frac{53}{54}$ . Using this and a reduction similar to the one presented in Theorem 10 it seems possible to show that PEU with 5 terminals is hard to approximate within a factor  $(1 + \frac{1}{2000})$ . The proof of Theorem 10 extends to give the next result.

**Theorem 11.** *PVU is APX-hard even with only 6 terminals.*

These results may seem to suggest that PEU and PVU have the same approximation guarantees. But the next theorem shows that PVU is significantly harder than PEU. We show this by a reduction from the set-cover packing problem (or domatic number problem). Given a bipartite graph  $G(V_1 \cup V_2, E)$ , a *set-cover* (of  $V_2$ ) is a subset  $S \subseteq V_1$  such that every vertex of  $V_2$  has a neighbor in  $S$ . A *set-cover packing* is a collection of pairwise disjoint set-covers of  $V_2$ . The goal is to find a packing of set-covers of maximum size. Feige et al. [6] show that, unless  $NP \subseteq DTIME(n^{\log \log n})$ , there is no  $(1 - \epsilon) \ln n$  approximation algorithm for set-cover packing, where  $n = |V_1| + |V_2|$ . We have the following theorem.

**Theorem 12.** *PVU cannot be approximated within ratio  $(1 - \epsilon) \log n$ , for any  $\epsilon > 0$ , unless  $NP \subseteq DTIME(n^{\log \log n})$ .*

On the other hand, we can obtain an  $O(\sqrt{n} \log n)$  algorithm for PVCU (which contains PVU as a special case). To do so, consider the fractional version of PVCU obtained by relaxing the integrality condition in the IP formulation. The separation problem for the dual of this LP is the minimum node-weighted Steiner tree problem. For this problem, Guha and Khuller [12] give an  $O(\log n)$  approximation algorithm. Using the following analog of Theorem 6 (or Theorem 4.1 in [14]) we obtain a polytime  $O(\log n)$  approximation for fractional PVCU.

**Lemma 3.** *There is an  $\alpha$ -approximation for fractional PVCU if and only if there is an  $\alpha$ -approximation for the minimum node weighted Steiner tree problem.*

**Remark:** Lemma 3 and the fact that the minimum node weighted Steiner tree problem is hard to approximate within  $O(\log k)$  (with  $k$  being the number of terminals) yields an alternative proof for the  $\Omega(\log k)$  hardness of PVCU.

The algorithm for PVCU is similar to the ones we presented for PECD and PVCD. That is, we apply randomized rounding to the solution of the fractional PVCU instance. Skipping the details, this yields the following:

**Theorem 13.** *Given an instance of PVCU and any  $\epsilon > 0$ , we can find in polynomial time a set of Steiner trees (satisfying the vertex capacity constraints) of size at least  $\Omega(\max\{\varphi_f/\sqrt{n} \log n, \min\{\varphi_f^2/n \log^2 n, \varphi_f/\log n\}\})$ , where  $\varphi_f$  is the optimal value of the instance of fractional PVCU.*

### 3.1 Packing Vertex-Disjoint Priority Steiner Trees

The priority Steiner problem has been studied by Charikar et al. [4]. Here, we study the problem of packing vertex-disjoint priority Steiner trees of undirected graphs. (One difference with the earlier work in [4] is that weights and priorities are associated with vertices rather than with edges). Consider an undirected graph  $G = (V, E)$  with a set of terminals  $T \subseteq V$ , one of which is distinguished as the root  $r$ , every vertex  $v$  has a nonnegative integer  $p_v$  as its priority, and every Steiner vertex  $v \in V - T$  has a positive capacity  $c_v$ . A *priority Steiner tree* is a Steiner tree such that for each terminal  $t \in T$  every Steiner vertex  $v$  on the  $r, t$  path has priority  $p_v \geq p_t$ . In the problem PVCU-priority (Packing Vertex Capacitated Undirected Priority Steiner Trees) the goal is to find a maximum set of priority Steiner trees obeying vertex capacities (i.e., for each Steiner vertex  $v \in V - T$  the number of trees containing  $v$  is  $\leq c_v$ ). The algorithm we presented for PVCU extends to PVCU-priority, giving roughly the same approximation guarantee.

**Theorem 14.** *Given an instance of PVCU-priority and any  $\epsilon > 0$ , we can find in polynomial time a set of priority Steiner trees (satisfying the vertex capacity constraints) of size at least  $\Omega(\max\{\varphi_f/n^{\frac{1+\epsilon}{2}}, \min\{\varphi_f^2/n^{1+\epsilon}, \varphi_f/n^{\epsilon/2}\}\})$ , where  $\varphi_f$  is the optimal value of the instance of fractional PVCU-priority.*

On the other hand, we prove an  $\Omega(n^{\frac{1}{3}-\epsilon})$  hardness result for PVCU-priority by adapting the proof of Theorem 4 (thus improving on our logarithmic hardness result for PVCU). The main difference from the proof of Theorem 4 is that we use instances of the Undir-Node-USF problem (*Undirected Node capacitated Unsplittable Flow*) – which is shown to be NP-complete in [9] – instead of instances of 2DIRPATH as the modules that are placed on the “grey boxes” in Figure 1.

**Theorem 15.** *Given an instance of PVCU-priority, it is NP-hard to approximate the solution within  $O(n^{\frac{1}{3}-\epsilon})$  for any  $\epsilon > 0$ .*



## References

1. S.Arora and C.Lund, *Hardness of approximations*, in *Approximation Algorithms for NP-hard Problems*, Dorit Hochbaum Ed., PWS Publishing, 1996.
2. A.Baveja and A.Srinivasan, *Approximation algorithms for disjoint paths and related routing and packing problems*, Mathematics of Operations Research 25:255-280, 2000. Earlier version in FOCS 1997.
3. M.Charikar, C.Chekuri, T.Cheung, Z.Dai, A.Goel, S.Guha, and M.Li, *Approximation algorithms for directed Steiner problem*, J. Algorithms 33(1):73-91, 1999. Earlier version in SODA 1998.
4. M.Charikar, J.Naor, and B.Schieber, *Resource optimization in QoS multicast routing of real-time multimedia*, Proc. 19th Annual IEEE INFOCOM (2000).
5. R.Diestel, *Graph Theory*, Springer, New York, NY, 2000.
6. U.Feige, M.Halldorsson, G.Kortsarz, and A.Srinivasan, *Approximating the domatic number*, Siam J. Computing 32(1):172-195, 2002. Earlier version in STOC 2000.
7. P.Floréen, P.Kaski, J.Kohonen, and P.Orponen, *Multicast time maximization in energy constrained wireless networks*, in Proc. 2003 Joint Workshop on Foundations of Mobile Computing (DIALM-POMC 2003), San Diego, CA, 2003.
8. A.Frank, T.Király, M.Kriesell, *On decomposing a hypergraph into  $k$  connected sub-hypergraphs*, Discrete Applied Mathematics 131(2):373-383, 2003.
9. V.Guruswami, S.Khanna, R.Rajaraman, B.Shepherd, and M.Yannakakis, *Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems*, J. Computer and System Sciences 67(3):473-496, 2003. Earlier version in STOC 1999.
10. S.Fortune, J.Hopcroft, and J.Wyllie, *The directed subgraph homeomorphism problem*, Theoretical Computer Science 10(2):111-121, 1980.
11. M.Grötschel, A.Martin, and R.Weismantel, *The Steiner tree packing problem in VLSI design*, Mathematical Programming 78:265-281, 1997.
12. S.Guha and S.Khuller, *Improved methods for approximating node weighted Steiner trees and connected dominating sets*, Information and Computation 150:57-74, 1999. Earlier version in FST&TCS 1998.
13. E.Hazan, S.Safra, and O.Schwartz, *On the hardness of approximating  $k$ -dimensional matching*, Electronic Colloquium on Computational Complexity, Rep. No. 20, 2003.
14. K.Jain, M.Mahdian, M.R.Salavatipour, *Packing Steiner trees*, Proc. SODA 2003.
15. V.Kann, *Maximum bounded 3-dimensional matching is MAX SNP-complete*, Information Processing Letters 37:27-35, 1991.
16. P.Kaski, *Packing Steiner trees with identical terminal sets*, Information Processing Letters 91(1):1-5, 2004.
17. S.G.Kolliopoulos and C.Stein, *Approximating disjoint-path problems using packing integer programs*, Mathematical Programming 99:63-87, 2004. Earlier version in Proc. IPCO 1998.
18. J.Kratochvil and Z.Tuza, *On the complexity of bicoloring clique hypergraphs of graphs*, J. Algorithms 45:40-54, 2002. Earlier version in SODA 2000.
19. M.Kriesell, *Edge-disjoint trees containing some given vertices in a graph*, J. Combinatorial Theory (B) 88:53-65, 2003.
20. L.C.Lau, *An approximate max-Steiner-tree-packing min-Steiner-cut theorem*, manuscript, 2004.
21. A.Martin and R.Weismantel, *Packing paths and Steiner trees: Routing of electronic circuits*, CWI Quarterly 6:185-204, 1993.



# Approximation Hardness of Dominating Set Problems

Miroslav Chlebík<sup>1</sup> and Janka Chlebíková<sup>2\*</sup>

<sup>1</sup> Max Planck Institute for Mathematics in the Sciences  
Inselstraße 22-26, D-04103 Leipzig, Germany  
`chlebik@mis.mpg.de`

<sup>2</sup> Faculty of Mathematics, Physics and Informatics  
Mlynská dolina, 842 48 Bratislava, Slovakia  
`chlebikj@dcs.fmph.uniba.sk`

**Abstract.** We study approximation hardness of the MINIMUM DOMINATING SET problem and its variants in undirected and directed graphs. We state the first explicit approximation lower bounds for various kinds of domination problems (connected, total, independent) in bounded degree graphs. For most of dominating set problems we prove asymptotically almost tight lower bounds. The results are applied to improve the lower bounds for other related problems such as the MAXIMUM INDUCED MATCHING problem and the MAXIMUM LEAF SPANNING TREE problem.

## 1 Introduction

A *dominating set* in a graph is a set of vertices such that every vertex in the graph is either in the set or adjacent to a vertex in it. The MINIMUM DOMINATING SET problem (MIN-DS) asks for a dominating set of minimum size. The variants of dominating set problems seek for a minimum dominating set with some additional properties, e.g., to be independent, or to induce a connected graph. These problems arise in a number of distributed network applications, where the problem is to locate the smallest number of centers in networks such that every vertex is nearby at least one center.

**Definitions.** A dominating set  $D$  in a graph  $G$  is an *independent dominating set* if the subgraph  $G_D$  of  $G$  induced by  $D$  has no edges;  $D$  is a *total dominating set* if  $G_D$  has no isolated vertices; and  $D$  is a *connected dominating set* if  $G_D$  is a connected graph. The corresponding problems MINIMUM INDEPENDENT DOMINATING SET (MIN-IDS), MINIMUM TOTAL DOMINATING SET (MIN-TDS), and MINIMUM CONNECTED DOMINATING SET (MIN-CDS) ask for an independent, total, and connected dominating set of minimum size, respectively. We use acronym  $B$  for the problem in  $B$ -bounded graphs (graphs with maximum degree at most  $B$ ). Let  $ds(G)$  stand for the minimum cardinality of a dominating set in  $G$ . Similarly, let  $ids(G)$ ,  $tds(G)$ , and  $cds(G)$ , stand for the corresponding minima

---

\* Partially supported by the Science and Technology Assistance Agency grant APVT-20-018902.

for MIN-IDS, MIN-TDS, and MIN-CDS for  $G$ , respectively. For definiteness, the corresponding optimal value is set to infinity if no feasible solution exists for  $G$ . That means,  $tds(G) < \infty$  iff  $G$  has no isolated vertices, and  $cds(G) < \infty$  iff  $G$  is connected. It is easy to see that  $ds(G) \leq ids(G)$ ,  $ds(G) \leq tds(G)$ , and  $ds(G) \leq cds(G)$ . Moreover,  $tds(G) \leq cds(G)$  unless  $ds(G) = 1$ .

In fact, dominating set problems are closely tied to the well-known MINIMUM SET COVER problem (MIN-SC). Let a set system  $\mathcal{G} = (\mathcal{U}, \mathcal{S})$  be given, where  $\mathcal{U}$  is a universe and  $\mathcal{S}$  is a collection of (nonempty) subsets of  $\mathcal{U}$  such that  $\cup \mathcal{S} = \mathcal{U}$ . Any subcollection  $\mathcal{S}' \subseteq \mathcal{S}$  such that  $\cup \mathcal{S}' = \mathcal{U}$  is termed a *set cover*. MIN-SC asks for a set cover of minimum cardinality, whose size is denoted by  $sc(\mathcal{G})$ .

An instance  $\mathcal{G} = (\mathcal{U}, \mathcal{S})$  of MIN-SC can be viewed as a hypergraph  $\mathcal{G}$  with vertices  $\mathcal{U}$  and hyperedges  $\mathcal{S}$ . For an element  $x \in \mathcal{U}$  let  $\deg(x)$  denote the number of sets in  $\mathcal{S}$  containing  $x$ ,  $\deg(\mathcal{G}) := \max_{x \in \mathcal{U}} \deg(x)$ , and let  $\Delta(\mathcal{G})$  denote the size of the largest set in  $\mathcal{S}$ . We will also consider the restriction of MIN-SC to instances  $\mathcal{G}$  with bounded both parameters  $\Delta(\mathcal{G}) \leq k$  and  $\deg(\mathcal{G}) \leq d$  denoted by  $(k, d)$ -MIN-SC. Hence,  $(k, \infty)$ -MIN-SC corresponds to  $k$ -MIN-SC, in which instances  $\mathcal{G}$  are restricted to those with  $\Delta(\mathcal{G}) \leq k$ .

For a hypergraph  $\mathcal{G} = (\mathcal{U}, \mathcal{S})$  define its *dual hypergraph*  $\tilde{\mathcal{G}} = (\mathcal{S}, \mathcal{U}_{\mathcal{G}})$  such that vertices in  $\tilde{\mathcal{G}}$  are hyperedges of  $\mathcal{G}$  and hyperedges  $\mathcal{U}_{\mathcal{G}} = \{x_{\mathcal{G}} : x \in \mathcal{U}\}$  correspond to vertices of  $\mathcal{G}$  in the following sense: given  $x \in \mathcal{U}$ ,  $x_{\mathcal{G}}$  contains all  $S \in \mathcal{S}$  such that  $x \in S$  in  $\mathcal{G}$ . In the context of hypergraphs, MIN-SC is the MINIMUM VERTEX COVER problem (MIN-VC) for the dual hypergraph. Clearly,  $\deg$  and  $\Delta$  are dual notions in the hypergraph duality. In fact,  $(k, d)$ -MIN-SC is the same problem as  $(d, k)$ -MIN-VC, but in its dual formulation.

MIN-SC can be approximated by a natural greedy algorithm that iteratively adds a set that covers the highest number of yet uncovered elements. It provides an  $\mathcal{H}_{\Delta}$ -approximation, where  $\mathcal{H}_i := \sum_{j=1}^i \frac{1}{j}$  is the  $i$ -th harmonic number. This factor has been improved to  $\mathcal{H}_{\Delta} - \frac{1}{2}$  [7]. Additionally, Feige [8] has shown that the approximation ratio of  $\ln n$  for MIN-SC is the best possible (as a function of  $n := |\mathcal{U}|$ , up to a lower order additive term) unless  $\text{NP} \subseteq \text{DTIME}(n^{O(\log \log n)})$ .

For any NP-hard optimization problem  $\Pi$  one can define its approximation thresholds  $t_P$  and  $t_{NP}$  as follows  $t_P = \inf\{c > 1 : \text{there is a polynomial } c\text{-approximation algorithm for } \Pi\}$  and  $t_{NP} = \sup\{c \geq 1 : \text{achieving approximation ratio } c \text{ for } \Pi \text{ is NP-hard}\}$ . For definiteness,  $\inf \emptyset := \infty$ . Hence  $t_P < \infty$  iff  $\Pi$  is in APX. Further,  $t_P = 1$  iff  $\Pi$  has PTAS. Clearly  $t_{NP} \leq t_P$  unless  $P = NP$ .

**Our contributions.** Due to a close relation of MIN-DS to MIN-SC almost tight approximability results are known for MIN-DS in general graphs. The best upper bound, which is logarithmic in maximum degree of the graph, almost matches the lower bound. In this contribution we investigate the approximability of the dominating set problem and its several variants in subclasses of general graphs, bounded degree, and directed graphs.

In Section 2 we observe that similar approximation hardness results as for MIN-DS in general graphs hold for MIN-DS, MIN-TDS, and MIN-CDS in split or, alternatively, in bipartite graphs. For  $B$ -bounded graphs we prove in Section 3 asymptotically tight lower bounds of  $\ln B$  for MIN-DS, MIN-TDS, and MIN-

CDS even in bipartite graphs. We present the lower bound for  $B$ -MIN-IDS that increases linearly with  $B$ , similar to the upper bound. All lower bounds summarized in the following table are new contributions of this paper and hold even in bipartite graphs, upper bounds are due to [1], [7], [10].

	$B$ -MIN-DS	$B$ -MIN-CDS	$B$ -MIN-TDS	$B$ -MIN-IDS
Low.	$\ln B - C \ln \ln B$	$\ln B - C \ln \ln B$	$\ln B - C \ln \ln B$	$\delta B$
Upp.	$\mathcal{H}_{B+1} - \frac{1}{2}$	$\mathcal{H}_B + 2$	$\mathcal{H}_B - \frac{1}{2}$	$B - \frac{B-1}{B^2+1}$

In Section 4 we introduce various kinds of reductions to achieve lower bounds for  $B$ -MIN-DS and  $B$ -MIN-IDS problems for small values of  $B$ . These lower bounds are summarized in the table below (\* means that lower bound is achieved even in bipartite graphs), and upper bounds follow from [1], [7].

Problem	3-MIN-DS	4-MIN-DS	3-MIN-IDS	4-MIN-IDS
Lower bound	$\frac{391}{390}^*$	$\frac{100}{99}$	$\frac{681}{680}$	$\frac{294}{293}^*$
Upper bound	$\frac{19}{12}$	$\frac{107}{60}$	2	$\frac{65}{17}$

Section 5 is devoted domination problems in directed graphs. We show that in directed graphs with indegree bounded by a constant  $B \geq 2$  the directed version of MIN-DS has simple  $(B + 1)$ -approximation algorithm, but it is NP-hard to approximate within any constant smaller than  $B - 1$  for  $B \geq 3$  (1.36 for  $B = 2$ ). In directed graphs with outdegree bounded by a constant  $B \geq 2$  we prove almost tight approximation lower bound of  $\ln B$  for directed version of MIN-DS. We also point out that the problem to decide of whether there exists a feasible solution for the MIN-IDS problem in directed graphs is NP-complete (even for bounded degree graphs).

In Section 6 we apply inapproximability results obtained for domination and covering problems to improve on approximation hardness results of two graph optimization problems. We improve the previous lower bound for 3-MAX-IM (MAXIMUM INDUCED MATCHING). Additionally, our lower bound for  $B$ -MAX-IM in  $B$ -regular graphs ( $B$  large) almost matches known linear upper bound (only APX-completeness was previously known with a lower bound very close to 1, even for large  $B$ ). We also establish the first explicit lower bound  $\frac{245}{244}$  for the MAXIMUM LEAF SPANNING TREE problem, and this hardness results applies also to bipartite graphs with all vertices but one of degree at most 5.

## 2 Dominating Set Problems in General Graphs

As it is already known, MIN-DS in general graphs has the same approximation hardness as MIN-SC. In this section we prove similar hardness results for other domination problems even in restricted graph classes. Let us start with the simple reduction between domination and set cover problems.

**DS-SC reduction.** Each vertex  $v \in V$  of a graph  $G$  will correspond to an element of  $\mathcal{U}$ , hence  $\mathcal{U} := V$ , and the collection  $\mathcal{S}$  will consist of the sets  $N_v \cup \{v\}$  for each vertex  $v \in V$  (resp., only  $N_v$  for MIN-TDS), where  $N_v$  is the set of all neighbors of  $v$ . This reduction exactly preserves feasibility of solutions: every dominating set in  $G$  (resp., total dominating set for a graph without isolated vertices) corresponds to a set cover in  $(\mathcal{U}, \mathcal{S})$  of the same size, and vice versa.

In this way we get polynomial time algorithm with the approximation ratio  $\mathcal{H}_{(\deg(G)+1)} - \frac{1}{2}$  for MIN-DS and with the approximation ratio  $\mathcal{H}_{\deg(G)} - \frac{1}{2}$  for MIN-TDS using results for MIN-SC [7], where  $\deg(G)$  denotes the maximum degree of  $G$ . For the MIN-CDS problem there is known polynomial time algorithm with the approximation ratio  $\mathcal{H}_{\deg(G)} + 2$  [10].

Now we recall two reductions in the opposite direction that we use to obtain hardness results for dominating set problems. For each instance  $(\mathcal{U}, \mathcal{S})$  of MIN-SC we assign the  $(\mathcal{U}, \mathcal{S})$ -bipartite graph with bipartition  $(\mathcal{U}, \mathcal{S})$  and edges between  $\mathcal{S}$  and all elements  $x \in \mathcal{U}$ , for each  $S \in \mathcal{S}$ .

**Split SC-DS reduction.** Given an instance  $\mathcal{G} = (\mathcal{U}, \mathcal{S})$  of MIN-SC, create first a  $(\mathcal{U}, \mathcal{S})$ -bipartite graph and then make a clique of all vertices of  $\mathcal{S}$ . This reduction transforms any set cover in  $(\mathcal{U}, \mathcal{S})$  to the corresponding dominating set of the same size in the resulting split graph  $G$ . It is not difficult to see that a dominating set of minimum size in  $G$  is achieved also among dominating sets which contains only vertices from  $\mathcal{S}$ : any dominating set  $D$  in  $G$  can be efficiently transformed to the one, say  $D'$ , with  $|D'| \leq |D|$  and  $D' \subseteq \mathcal{S}$ . Since a dominating set contained in  $\mathcal{S}$  induces a clique, problems MIN-CDS, MIN-TDS, and MIN-DS have the same complexity in graphs constructed using split SC-DS reduction.

**Bipartite SC-DS reduction.** Given an instance  $\mathcal{G} = (\mathcal{U}, \mathcal{S})$  of MIN-SC, create first a  $(\mathcal{U}, \mathcal{S})$ -bipartite graph. Then add two new vertices  $y$  and  $y'$ , and connect  $y$  to each  $S \in \mathcal{S}$  and to  $y'$ . For the resulting bipartite graph  $G$  one can now confine to dominating sets consisting of  $y$  and a subset of  $\mathcal{S}$  corresponding to a set cover, hence we have  $ds(G) = cds(G) = tds(G) = sc(\mathcal{G}) + 1$ .

In order to transfer Feige's  $(1 - \varepsilon) \ln |\mathcal{U}|$  hardness result [8] from MIN-SC to dominating set problems using both SC-DS reductions, we need to know that “hard instances” of MIN-SC satisfy  $\ln(|\mathcal{U}| + |\mathcal{S}|) \approx \ln(|\mathcal{U}|)$ . Analyzing Feige's construction we turn out that this is indeed true. Hence known hardness results for MIN-DS hold also for MIN-TDS and MIN-CDS even in split and bipartite graphs.

**Theorem 1.** *MIN-DS, MIN-TDS, and MIN-CDS cannot be approximated to within a factor of  $(1 - \varepsilon) \ln n$  in polynomial time for any constant  $\varepsilon > 0$ , unless  $\text{NP} \subseteq \text{DTIME}(n^{O(\log \log n)})$ . The same results hold also in split graphs (and hence in chordal graphs, and in complements of chordal graphs as well), and in bipartite graphs.*

The MIN-IDS problem is NP-hard, and in fact, no method to approximate  $ids$  within a factor better than trivial one  $O(n)$  appears to be known. Halldórsson [11] proved that MIN-IDS cannot be approximated in polynomial time within a factor of  $n^{1-\varepsilon}$  for any  $\varepsilon > 0$ , unless  $\text{P} = \text{NP}$ .

### 3 Dominating Set Problems in Bounded Degree Graphs

Now we consider domination set problems in graphs of maximum degree  $B$ . Due to results in general graphs, we have  $t_P(B\text{-MIN-DS}) \leq \mathcal{H}_{B+1} - \frac{1}{2}$ ,  $t_P(B\text{-MIN-TDS}) \leq \mathcal{H}_B - \frac{1}{2}$ , and  $t_P(B\text{-MIN-CDS}) \leq \mathcal{H}_B + 2$ . In what follows we prove asymptotically tight lower bounds of  $\ln B$  (up to lower order terms) for all three mentioned problems.

#### $B$ -MINIMUM DOMINATING SET

Trevisan [16] in the analysis of Feige's construction proved the following inapproximability result for  $B$ -MIN-SC.

**Theorem 2 (Trevisan).** *There are absolute constants  $C > 0$  and  $B_0 \geq 3$  such that for every  $B \geq B_0$  it is NP-hard to approximate the  $B$ -MIN-SC problem within a factor of  $\ln B - C \ln \ln B$ .*

In fact, in the proof of the corresponding NP-hard gap type result an instance  $\Psi$  of SAT of size  $n$  is converted to an instance  $\mathcal{G} = (\mathcal{U}, \mathcal{S})$  of  $B$ -MIN-SC. There are parameters  $l$  and  $m$ ,  $l$  is fixed to  $\theta(\ln \ln B)$  and  $m$  is fixed to  $\frac{B}{\text{poly log } B}$ . The produced instances have the following properties:  $|\mathcal{U}| = mn^l \text{poly log } B$ ,  $|\mathcal{S}| = n^l \text{poly log } B$ ,  $\Delta(\mathcal{G}) \leq B$ , and  $\deg(\mathcal{G}) \leq \text{poly log } B$ . Further, if  $\Psi$  is satisfiable then  $sc(\mathcal{G}) < \alpha|\mathcal{S}|$  (for some  $\alpha$  easily computable from  $n$  and  $B$ ), but if  $\Psi$  is not satisfiable, then  $sc(\mathcal{G}) > \alpha|\mathcal{S}|(\ln B - C \ln \ln B)$ .

We use Trevisan's result to prove inapproximability results for  $B$ -MIN-DS for large  $B$ . In such case the following reduction from  $(B-1, B)$ -MIN-SC to  $B$ -MIN-DS can be used as a gap preserving reduction.

**SC-DS<sub>1</sub> reduction.** Let  $\mathcal{G} = (\mathcal{U}, \mathcal{S})$  be an instance of  $(B-1, B)$ -MIN-SC and  $(\mathcal{U}, \mathcal{S})$ -bipartite corresponding graph. Add a set  $W$  of  $\left\lceil \frac{|\mathcal{S}|}{B} \right\rceil$  new vertices connected to the  $(\mathcal{U}, \mathcal{S})$ -bipartite graph as follows: each vertex  $S \in \mathcal{S}$  is connected to one vertex of  $W$  and allocate these edges to vertices of  $W$  such that degree of each vertex in  $W$  is also at most  $B$ . Let  $G$  denote the bipartite graph of degree at most  $B$  constructed in this way. It can be proved that the SC-DS<sub>1</sub> reduction has the properties:  $sc(\mathcal{G}) \leq ds(G) \leq sc(\mathcal{G}) + \left\lceil \frac{|\mathcal{S}|}{B} \right\rceil$ .

The SC-DS<sub>1</sub> reduction translates the NP-hard question for  $(B-1, B)$ -MIN-SC to decide of whether  $sc(\mathcal{G}) < \alpha|\mathcal{S}|$ , or  $sc(\mathcal{G}) > \beta|\mathcal{S}|$  (for some efficiently computable functions  $\alpha, \beta$ ) to the NP-hard question of whether  $ds(G) < (\alpha + \frac{1}{B})|\mathcal{S}|$ , or  $ds(G) > \beta|\mathcal{S}|$  (assuming  $\beta - \alpha > \frac{1}{B}$ ).

It is easy to check that the SC-DS<sub>1</sub> reduction from  $(B-1, \text{poly log } B)$ -MIN-SC to  $B$ -MIN-DS can decrease the approximation hardness factor of  $\ln(B-1) - C \ln \ln(B-1)$  from Theorem 2 only marginally (by an additive term of  $\frac{\text{poly log } B}{B}$ ). Hence an approximation threshold  $t_{NP}$  for  $B$ -MIN-DS and  $B$  sufficiently large is again at least  $\ln B - C \ln \ln B$ , with slightly larger constant  $C$  than in Theorem 2. Hence we obtain the following

**Theorem 3.** *There are absolute constants  $C > 0$  and  $B_0 \geq 3$  such that for every  $B \geq B_0$  it is NP-hard to approximate  $B$ -MIN-DS (even in bipartite graphs) within a factor of  $\ln B - C \ln \ln B$ .*

$B$ -MIN TOTAL DOMINATING SET and  $B$ -MIN CONNECTED DOMINATING SET  
 We modify slightly the SC-DS<sub>1</sub> reduction for  $B$ -MIN-TDS and  $B$ -MIN-CDS.

**SC-DS<sub>2</sub> reduction.** For an instance  $\mathcal{G} = (\mathcal{U}, \mathcal{S})$  of  $(B-1, B)$ -MIN-SC (with  $B$  sufficiently large) construct the  $(\mathcal{U}, \mathcal{S})$ -bipartite graph and add a set  $W$  of  $\left\lceil \frac{|\mathcal{S}|}{B-2} \right\rceil$  new vertices. Connect them to vertices of  $\mathcal{S}$  in the same way as in SC-DS<sub>1</sub> reduction and add a set  $W'$  of additional vertices,  $|W'| = |W|$ . The vertices of  $W$  and  $W'$  are connected to a  $2|W|$ -cycle with vertices of  $W$  and  $W'$  alternating in it. The result of this transformation will be a bipartite graph  $G$  of maximum degree at most  $B$  with the following properties  $sc(\mathcal{G}) \leq tds(G) \leq cds(G) \leq sc(\mathcal{G}) + 2 \left\lceil \frac{|\mathcal{S}|}{B-2} \right\rceil$ . Hence we obtain essentially the same asymptotical results as for  $B$ -MIN-DS.

**Theorem 4.** *There are absolute constants  $C > 0$  and  $B_0 \geq 3$  such that for every  $B \geq B_0$  it is NP-hard to approximate (even in bipartite graphs)  $B$ -MIN-TDS, resp.  $B$ -MIN-CDS, within a factor of  $\ln B - C \ln \ln B$ .*

#### $B$ -MINIMUM INDEPENDENT DOMINATING SET

One can easily observe that for any  $(B+1)$ -claw free graph  $G$  we have  $|I| \leq B|D|$  whenever  $I$  is an independent set and  $D$  is a dominating set in  $G$ . Consequently, any independent dominating set in  $(B+1)$ -claw free graph  $G$  approximates MIN-IDS, MIN-DS, and MAX-IS (MAXIMUM INDEPENDENT SET) within  $B$ . It trivially applies to  $B$ -bounded graphs as well. For many problems significantly better approximation ratios are known for  $B$ -bounded graphs than for  $(B+1)$ -claw free graphs. For the  $B$ -MIN-IDS problem (asymptotically) only slightly better upper bounds are known [1], namely  $t_P \leq B - \frac{B-1}{B^2+1}$  for  $B \geq 4$  and  $t_P \leq 2$  for  $B = 3$ . (For  $B$ -regular instances,  $B \geq 5$ ,  $t_P \leq B - 1 - \frac{B-3}{B^2+1}$ .)

The question of whether there are polynomial time algorithms for  $B$ -MIN-IDS with ratios  $o(B)$  when  $B$  approaches infinity is answered in the negative (unless  $P = NP$ ) by the following theorem. Our proof extracts the core of arguments used in [11] and [13]. Starting from NP-hard gap result for bounded occurrence version of MAX-3SAT, we obtain NP-hard gap type result for  $B$ -MIN-IDS, for large  $B$ .

**Theorem 5.** *There are absolute constants  $\delta > 0$  and  $B_0$  such that for every  $B \geq B_0$  the  $B$ -MIN-IDS problem is NP-hard to approximate within  $\delta B$ . The same hardness result applies to bipartite graphs as well.*

## 4 Dominating Set Problems in Small Degree Graphs

Graphs of degree at most 2 have simple structure and all problems studied above can be solved efficiently in this class. Thus suppose that  $B \geq 3$ .

#### $B$ -MINIMUM DOMINATING SET

Using the standard DS-SC reduction and results for  $(B+1)$ -MIN-SC [7] we obtain upper bounds  $t_P(3\text{-MIN-DS}) \leq \frac{19}{12}$ ,  $t_P(4\text{-MIN-DS}) \leq \frac{107}{60}$ ,  $t_P(5\text{-MIN-DS}) \leq$

$\frac{117}{60}$ , and  $t_P(6\text{-MIN-DS}) \leq \frac{879}{420}$ , respectively. To obtain a lower bound on approximability for  $B\text{-MIN-DS}$  from bounded version of  $\text{MIN-SC}$  we cannot rely on the split and bipartite  $\text{SC-DS}$  reductions, as only finitely many instances of  $\text{MIN-SC}$  will transform to instances of  $B\text{-MIN-DS}$  by those reductions. Instead of that we can use the following simple  $\text{VC-DS}$  reduction  $f$  from  $\text{MIN-VC}$  to  $\text{MIN-DS}$  to obtain a lower bound on  $t_{\text{NP}}$  for  $B\text{-MIN-DS}$  for small values of  $B$ .

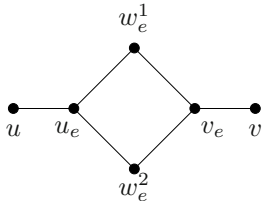


Fig. 1.

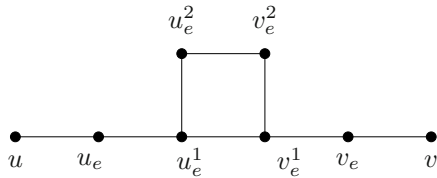


Fig. 2.

**VC-DS reduction.** Given a graph  $G = (V, E)$  with  $n$  vertices and  $m$  edges (without isolated vertices), replace each edge  $e = \{u, v\} \in E$  by a gadget  $G_e$  (Fig. 1). The constructed graph  $f(G)$  has  $n + 4m$  vertices and  $6m$  edges. Moreover,  $f(G)$  is bipartite, and if  $G$  is of degree  $B$  ( $\geq 3$ ) then the same is true for  $f(G)$ . Furthermore, the  $\text{VC-DS}$  reduction has the property  $ds(f(G)) = vc(G) + m$ , where  $vc(G)$  denote the minimum cardinality of a vertex cover in  $G$ .

Applying the  $\text{VC-DS}$  reduction to a 3-regular graph  $G$  with  $n$  vertices produces a bipartite graph  $f(G)$  of maximum degree 3 with  $7n$  vertices and  $9n$  edges. Using NP-hard gap result for 3-MIN-VC [3] we obtain that it is NP-hard to decide of whether  $ds(f(G))$  is greater than  $2.01549586n$ , or less than  $2.0103305n$ , hence to approximate  $\text{MIN-DS}$  within  $\frac{391}{390}$  is NP-hard.

**Theorem 6.** *It is NP-hard to approximate (even in bipartite graphs) 3-MIN-DS within  $1 + \frac{1}{390}$ .*

For larger value of  $B$ ,  $B \geq 4$ , better inapproximability results can be achieved by the following  $\text{SC-DS}_3$  reduction.

**SC-DS<sub>3</sub> reduction.** From an instance  $\mathcal{G} = (\mathcal{U}, \mathcal{S})$  of  $\text{MIN-SC}$  we construct firstly the  $(\mathcal{U}, \mathcal{S})$ -bipartite graph. Then for each  $S \in \mathcal{S}$  we pick one fixed representative  $u_S \in S$  and add new edges to the  $(\mathcal{U}, \mathcal{S})$ -bipartite graph connecting  $S$  with each other  $S' \in \mathcal{S}$  containing  $u_S$ . (We avoid creating multiple edges). Let  $G$  denote the resulting graph. It can be proved that the  $\text{SC-DS}_3$  reduction has the property  $ds(G) = sc(\mathcal{G})$ .

In what follows we prove that the  $\text{SC-DS}_3$  reduction can be modified to the one from  $(B - 1)\text{-MIN-VC}$  (with a perfect matching) to  $B\text{-MIN-DS}$ , that preserves the optimum of the objective function. Let  $H = (V, E)$  be an instance of  $(B - 1)\text{-MIN-VC}$  with a fixed perfect matching  $M$  in it. Let  $\tilde{\mathcal{G}} = (E, V_H)$  be the dual hypergraph to (hyper)graph  $H$ . Due to duality,  $\tilde{\mathcal{G}}$  can be viewed as  $(B - 1, 2)$ -instance of  $\text{MIN-SC}$  and  $sc(\tilde{\mathcal{G}}) = vc(H)$ . The corresponding  $(E, V_H)$ -bipartite graph for  $\tilde{\mathcal{G}}$  is just division of  $H$  (for every edge  $e$  put a single vertex



on it, labeled by  $e$ ), if one identifies each  $v \in V$  with the corresponding set  $v_H$  containing all edges incident with  $v$  in  $H$ . Now we consider SC-DS<sub>3</sub> reduction and for each set  $S$  (corresponding to  $v \in V$ ) we take as  $u_S$  exactly that edge adjacent to  $v$  in  $H$  that belongs to  $M$ . Hence the resulting graph  $G$  can be obtained from a division of  $H$  by adding edges of  $M$ . Therefore,  $G$  is of degree at most  $B$  and  $ds(G) = sc(\tilde{G}) = vc(H)$ .

It is easy to verify that NP-hard gap results for  $B$ -MIN-VC [3] ( $B = 3, 4$ , and  $5$ ) apply to  $B$ -regular graphs with a perfect matching as well. (For  $B = 3, 4$  it is proved in [3] that hard instances produced there are  $B$ -regular and edge  $B$ -colorable.) It means for  $B = 4, 5$ , and  $6$  we obtain for  $B$ -MIN-DS the same lower bound as for  $(B - 1)$ -MIN-VC.

**Theorem 7.** *It is NP-hard to approximate 4-MIN-DS within  $1 + \frac{1}{99}$ , 5-MIN-DS within  $1 + \frac{1}{52}$ , and 6-MIN-DS within  $1 + \frac{1}{50}$ .*

*Remark 1.* From results [2] for MINIMUM EDGE DOMINATING SET it also follows that for 4-regular graphs obtained as line graphs of 3-regular graphs it is NP-hard to approximate MIN-DS within  $1 + \frac{1}{390}$ . Recall that for MIN-DS restricted to line graphs there is a simple 2-approximation algorithm, but it is NP-hard to approximate within a constant smaller than  $\frac{7}{6}$  by results of [2].

#### $B$ -MINIMUM INDEPENDENT DOMINATING SET

As it is proved in [1]  $t_P(3\text{-MIN-IDS}) \leq 2$  and the upper bound  $t_P(B\text{-MIN-IDS}) \leq B - \frac{B-1}{B^2+1}$  holds for any small value  $B \geq 4$ . It means  $t_P(4\text{-MIN-IDS}) \leq \frac{65}{17}$ ,  $t_P(5\text{-MIN-IDS}) \leq \frac{63}{13}$ , and  $t_P(6\text{-MIN-IDS}) \leq \frac{217}{37}$ . To obtain inapproximability results for  $B$ -MIN-IDS for small values of  $B \geq 3$ , we use the following polynomial time reduction from MIN-SC.

**SC-IDS reduction.** Let an instance  $\mathcal{G} = (\mathcal{U}, \mathcal{S})$  of  $(B - 1, B)$ -MIN-SC be given. Start with the corresponding  $(\mathcal{U}, \mathcal{S})$ -bipartite graph and for each  $S \in \mathcal{S}$  add two new vertices  $S', S''$  and two edges  $\{S, S'\}, \{S', S''\}$ . The resulting graph  $G$  is bipartite of maximum degree at most  $B$  and  $ids(G) = ds(G) = sc(\mathcal{G}) + |\mathcal{S}|$ .

Using the previous we can obtain NP-hard gap result for  $B$ -MIN-IDS (and  $B$ -MIN-DS as well) from the one for  $(B - 1, B)$ -MIN-SC, or equivalently, for the hypergraph  $(B, B - 1)$ -MIN-VC problem whose special case is the  $(B - 1)$ -MIN-VC problem in graphs. More precisely, we can translate NP-hard gap results of [3] for  $(B - 1)$ -MIN-VC to the ones for  $B$ -MIN-IDS as follows: Starting from a 3-regular instance for 3-MIN-VC with  $n$  vertices using SC-IDS reduction we obtain a bipartite graph  $G$  of degree at most 4 and with the NP-hard question of whether  $ids(G)$  is greater than  $1.51549586n$  or less than  $1.5103305n$ . Hence, it is NP-hard to approximate 4-MIN-IDS even in bipartite graphs within  $\frac{294}{293}$ . In the same way we can obtain inapproximability results for 5(6)-MIN-IDS starting from 4 and 5-regular graphs, respectively.

**Theorem 8.** *It is NP-hard to approximate (even in bipartite graphs) 4-MIN-IDS within  $1 + \frac{1}{293}$ , 5-MIN-IDS within  $1 + \frac{1}{151}$ , and 6-MIN-IDS within  $1 + \frac{1}{145}$ .*

To obtain a lower bound for 3-MIN-IDS, let us consider the following reduction  $h$  from MIN-VC to MIN-IDS:



**VC-IDS reduction.** Given a graph  $G = (V, E)$  with  $n$  vertices and  $m$  edges (without isolated vertices), replace each edge  $e = \{u, v\} \in E$  by a simple edge gadget  $G_e$  (Fig. 2). The graph  $h(G)$  constructed in this way has  $n + 6m$  vertices and  $8m$  edges. Moreover, if  $G$  is of maximum degree  $B$  ( $\geq 3$ ) then the same is true for  $h(G)$  and  $\text{ids}(h(G)) = \text{vc}(G) + 2m$ .

Applying the VC-IDS reduction to a instance  $G$  of 3-MIN-VC (with  $n$  vertices) and using NP-hard gap result for it [2], we obtain that it is NP-hard to decide of whether  $\text{ids}(h(G))$  is greater than  $3.51549586n$ , or less than  $3.5103305n$ . Hence to approximate 3-MIN-IDS within  $\frac{681}{680}$  is NP-hard.

**Theorem 9.** *It is NP-hard to approximate 3-MIN-IDS within  $1 + \frac{1}{680}$ .*

## 5 Minimum Dominating Set in Directed Graphs

In a directed graph  $G = (V, \vec{E})$  a set  $D \subseteq V$  is a *dominating set* if for each  $v \in V \setminus D$  there is  $u \in D$  such that  $\vec{uv} \in \vec{E}$ . Again, it is special case of MIN-SC due to the following directed DS-SC reduction:

**Directed DS-SC reduction.** For a directed graph  $G = (V, \vec{E})$ , we define an instance  $(\mathcal{U}, \mathcal{S})$  of MIN-SC as  $\mathcal{U} := V$  and  $\mathcal{S} := \{N_v^+ : v \in V\}$ , where  $N_v^+ = \{v\} \cup \{u \in V : \vec{vu} \in \vec{E}\}$ . For such instance  $(\mathcal{U}, \mathcal{S})$ , set covers are in one-to-one correspondence with dominating sets in  $G$ .

### Minimum Dominating Set in Graphs with Bounded Indegree

Due to the directed DS-SC reduction, instances of MIN-DS with indegree bounded by a constant  $B$  can be viewed as instances  $\mathcal{G}$  of MIN-SC with  $\deg(\mathcal{G}) \leq B + 1$ . Hence the problem has a simple  $(B + 1)$ -approximation algorithm in this case. Furthermore, case  $B = 1$  can be easily solved exactly.

Asymptotically, we can obtain almost matching lower bound by the following reduction from restricted instances of MIN-SC to directed instances of MIN-DS: for an instance  $\mathcal{G} = (\mathcal{U}, \mathcal{S})$  with  $\deg(\mathcal{G}) \leq B$  construct a graph  $G$  with the vertex set  $V = \mathcal{U} \cup \mathcal{S} \cup \{S_0\}$ , where  $S_0$  is a new vertex. Add edges  $\vec{S_0 S}$  in  $G$  for each  $S \in \mathcal{S}$ , and an edge  $\vec{Sx}$  for each  $S \in \mathcal{S}$  and each  $x \in S$ . The directed graph  $G = (V, \vec{E})$  created in this way has indegree bounded by  $B$ . Obviously, there are minimum dominating sets in  $G$  consisting of  $S_0$  and  $\mathcal{C} \subseteq \mathcal{S}$ , where  $\mathcal{C}$  is a minimum set cover in  $(\mathcal{U}, \mathcal{S})$ . Hence this reduction preserves NP-hard gap results for MIN-SC restricted to instances  $\mathcal{G}$  with  $\deg(\mathcal{G}) \leq B$ . Recall that this is equivalent to the MIN-VC problem in hypergraphs with hyperedges of size at most  $B$ . Recently Dinur et al. ([4]) gave nearly tight lower bound  $(B - 1)$  on approximability in hypergraphs with all hyperedges of size exactly  $B$ ,  $B \geq 3$ . For  $B = 2$  the lower bound 1.36 follows from the one for MIN-VC in graphs [5].

**Theorem 10.** *It is NP-hard to approximate the MIN-DS problem in directed graphs with indegree bounded by a constant  $B$  within any constant smaller than  $B - 1$  for  $B \geq 3$ , and within 1.36 for  $B = 2$ . On the other hand, the problem has a simple  $(B + 1)$ -approximation algorithm for  $B \geq 2$ .*



### Minimum Dominating Set in Graphs with Bounded Outdegree

Instances of MIN-DS with outdegree bounded by a constant  $B$  can be viewed as instances of MIN-SC with sets of size at most  $B + 1$ . Hence the problem is polynomially solvable for  $B = 1$ , and for  $B \geq 2$  polynomial time approximation algorithm with the ratio  $\mathcal{H}_{B+1} - \frac{1}{2} < \ln B + O(1)$  is known [7].

To obtain some lower bounds for MIN-DS in graphs with outdegree bounded by a constant  $B$  we consider case when both, outdegree and indegree are bounded by  $B$ . This case is at least as hard as non-directed  $B$ -MIN-DS: replace in instances of  $B$ -MIN-DS every edge  $\{u, v\}$  by two directed edges  $\overrightarrow{uv}, \overrightarrow{vu}$ , and observe that this reduction preserves dominating sets. Hence directly from Theorem 3 we can obtain the following theorem

**Theorem 11.** *There are absolute constants  $C > 0$  and  $B_0 \geq 3$  such that for every  $B \geq B_0$  it is NP-hard to approximate MIN-DS in directed graphs with outdegree bounded by a constant  $B$  within  $\ln B - C \ln \ln B$ . However, there exists  $(\mathcal{H}_{B+1} - \frac{1}{2})$ -approximation algorithm for the problem for any  $B, B \geq 2$ .*

### Other Dominating Set Problems in Directed Graphs

The variants of MIN-DS, namely MIN-TDS, MIN-CDS and MIN-IDS, can be formulated for directed graphs as well. Mainly for connected domination problems there are many interesting questions left open. MIN-IDS in directed graphs is very different from its undirected counterpart. The problem to decide of whether there exists a feasible solution in an input directed graph is NP-complete even in bounded degree graphs due to the following reduction from MAX-3SAT-5: given an instance  $\phi$ , create a graph  $G_\phi$  with two vertices labeled by  $x$  and  $\bar{x}$ , for every variable  $x$ , and three vertices labeled by  $c, c',$  and  $c''$ , for every clause  $C$ . Edges are chosen so that every pair  $x, \bar{x}$  is a 2-cycle , every triple  $c, c', c''$  is a directed 3-cycle , and there is an edge  $\overrightarrow{lc}$  whenever literal  $l$  is in a clause  $C$ . One can easily check that  $G_\phi$  has an independent dominating set iff  $\phi$  is satisfiable. Moreover,  $G_\phi$  has fulldegree bounded by 7.

## 6 Application to Other Problems

### MAXIMUM INDUCED MATCHING PROBLEM (MAX-IM)

A matching  $M$  in a graph  $G = (V, E)$  is *induced* if for each edge  $e = \{u, v\} \in E$ ,  $u, v \in V(M)$  implies  $e \in M$ . The objective of MAX-IM is to find a maximum induced matching in  $G$ , let  $im(G)$  denote its cardinality.

The problem is known to be NP-complete even in bipartite graphs of maximum degree 3 and the current state of the art can be found in [6], [14], and [17]. For  $B$ -MAX-IM,  $B \geq 3$ , any inclusionwise maximal induced matching approximates the optimal solution within  $2(B - 1)$  (see [17]). This was improved to an asymptotic ratio  $B - 1$  in  $B$ -regular graphs in [6], where also the proof of APX-completeness of  $B$ -MAX-IM in  $B$ -regular graphs is given.

In what follows we present a lower bound for  $B$ -MAX-IM in  $B$ -regular graphs (for large  $B$ ) that approaches infinity with  $B$  and almost matches linear upper bound. Firstly, one can easily check that the lower bound of  $\frac{B}{2^{O(\sqrt{\ln B})}}$  given by

Trevisan [16] for  $B$ -MAX-IS applies to  $B$ -regular graphs as well. Now consider the following transformation  $g$  for a  $(B - 1)$ -regular graph  $G = (V, E)$ : take another copy  $G' = (V', E')$  of the same graph  $G$  (with  $v' \in V'$  corresponding to  $v \in V$ ), and make every pair  $\{v, v'\}$  adjacent. The resulting graph is  $B$ -regular and it is easy to observe that  $is(G) \leq im(g(G)) \leq 2is(G)$ . Hence a lower bound on approximability for  $B$ -MAX-IM in  $B$ -regular graphs is at least  $\frac{1}{2}$  of Trevisan's one, it means again of the form  $\frac{B}{2^{O(\sqrt{\ln B})}}$ .

For any  $B \geq 4$  we can use the following simple reduction  $f$  from  $(B - 1)$ -MAX-IS to  $B$ -MAX-IM:  $f(G)$  is constructed from a graph  $G$  adding a pending  $\{v, v'\}$  at each vertex  $v$  of  $G$ . Obviously,  $im(f(G)) = is(G)$  and hence NP-hard gap results for  $(B - 1)$ -MAX-IS directly translates to the one for  $B$ -MAX-IM. In particular,  $t_{NP}(4\text{-MAX-IM}) > \frac{95}{94}$ ,  $t_{NP}(5\text{-MAX-IM}) > \frac{48}{47}$ , and  $t_{NP}(6\text{-MAX-IM}) > \frac{46}{45}$ .

The problem to obtain any decent lower bound for 3-MAX-IM is more difficult. One can observe (e.g., [14]) that for any graph  $G = (V, E)$  its subdivision  $G^0$  (replace every edge  $\{u, v\}$  in  $G$  with a path  $u, w, v$  through a new vertex  $w$ ) satisfies  $im(G^0) = |V| - ds(G)$ . Using NP-hard gap result for 3-MIN-DS from Theorem 6, we obtain instances  $G^0$  of maximum degree 3 with  $16n$  vertices,  $18n$  edges, and with the NP-hard question to decide if whether  $im(G^0)$  is greater than  $4.9896695n$ , or less than  $4.9845042n$ . Hence to approximate 3-MAX-IM even in subdivision (and, in particular, bipartite) graphs within  $\frac{967}{966}$  is NP-hard. It improves the previous lower bound  $\frac{6600}{6659}$  for the 3-MAX-IM problem in bipartite graphs from [6]. Using the reduction from MAX-IS to MAX-IM presented in [6], we can improve also a lower bound  $\frac{475}{474}$  for 3-MAX-IM in general graphs. From a 3-regular instance  $G$  of 3-MAX-IS with  $n$  vertices, in the combination with NP-hard gap results for them ([3]), we produce an instance  $G'$  of 3-MAX-IM (with  $5n$  vertices,  $\frac{11}{2}n$  edges, and with  $im(G') = n + is(G)$ ) with the NP-hard question to decide if whether  $im$  is greater than  $1.51549586n$  or less than  $1.5103305n$ .

**Theorem 12.** *It is NP-hard to approximate 3-MAX-IM within  $1 + \frac{1}{293}$ , and within  $1 + \frac{1}{966}$  in graphs that are additionally bipartite. Further, it is NP-hard to approximate 4-MAX-IM within  $1 + \frac{1}{94}$ , 5-MAX-IM within  $1 + \frac{1}{47}$ , and 6-MAX-IM within  $1 + \frac{1}{45}$ . Asymptotically, it is NP-hard to approximate  $B$ -MAX-IM within a factor  $\frac{B}{2^{O(\sqrt{\ln B})}}$  (in  $B$ -regular graphs as well).*

#### MAXIMUM LEAF SPANNING TREE PROBLEM (MAX-LST)

The goal of MAX-LST is for a given connected graph to find a spanning tree with the maximum number of leaves. The problem is approximable within 3 [15] and known to be APX-complete [9].

If  $G = (V, E)$  is a connected graph with  $|V| \geq 3$  then  $|V| - cds(G)$  is the maximum number of leaves in a spanning tree of  $G$ . This simple observation allows us to obtain the first explicit inapproximability results for MAX-LST. The NP-hard gap result for 4-MIN-VC in 4-regular graphs [3] implies the same NP-hard gap for the  $(4, 2)$ -MIN-SC problem due to the duality of both problems. Hence it is NP-hard to decide if the optimum for  $(4, 2)$ -MIN-SC is greater than  $0.5303643725n$  or smaller than  $0.520242915n$ , where  $n$  is the number of vertices for dual 4-regular graph. Applying the bipartite SC-DS reduction for such hard

instances of  $(4, 2)$ -MIN-SC we obtain a bipartite graph with  $3n + 2$  vertices, all but one of degree at most 5, and with the NP-hard question for MAX-LST to decide if whether the optimum is less than  $2.469635627n + 1$ , or greater than  $2.479757085n + 1$ . Hence inapproximability within  $\frac{245}{244}$  follows.

**Theorem 13.** *It is NP-hard to approximate (even in bipartite graphs with all vertices but one of degree at most 5) MAX-LST within  $1 + \frac{1}{244}$ .*

## References

1. P. Alimonti and T. Calamoneri. Improved approximations of independent dominating set in bounded degree graphs. In *Proc. of the 22nd International Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS, pp. 2–16, 1996.
2. M. Chlebík and J. Chlebíková. Approximation hardness of minimum edge dominating set and minimum maximal matching. In *Proc. of the 14th International Symp. on Algorithms and Computation 2003*, LNCS 2906, pp. 415–424, 2003.
3. M. Chlebík and J. Chlebíková. Inapproximability results for bounded variants of optimization problems. In *Proc. of the 14th International Symposium on Fundamentals of Computation Theory*, LNCS 2751, pp. 27–38, 2003.
4. I. Dinur, V. Guruswami, S. Khot, and O. Regev. A new multilayered PCP and the hardness of hypergraph vertex cover. In *Proc. of the 35th ACM Symposium on Theory of Computing*, 2003, pp. 595–601.
5. I. Dinur and S. Safra. The importance of being biased. In *Proc. of the 34th ACM Symposium on Theory of Computing*, 2002, pp. 33–42.
6. W. Duckworth, D. F. Manlove, and M. Zito. On the approximability of the maximum induced matching problem. to appear in *Journal of Discrete Algorithms*.
7. R. Duh and M. Fürer. Approximation of  $k$ -set cover by semi-local optimization. In *Proc. of the 29th ACM Symp. on Theory of Computing 1997*, pp. 256–264.
8. U. Feige. A threshold of  $\ln n$  for approximation set cover. *Journal of ACM*, 45(4):634–652, 1998.
9. G. Galbiati, F. Maffioli, and A. Morzenti. A short note on the approximability of the maximum leaves spanning tree problem. *Information Processing Letters*, 52:45–49, 1994.
10. S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. In *Proc. of the 4th Ann. European Symp. on Alg.*, LNCS 1136, pp. 179–193, 1996.
11. M. M. Halldórsson. Approximating the minimum maximal independence number. *Information Processing Letters*, 46:169–172, 1993.
12. J. Håstad. Some optimal inapproximability results. *Journal of ACM*, 48(4):798–859, 2001.
13. H. W. Irving. On approximating the minimum independent dominating set. *Inform. Process. Lett.*, 37:197–200, 1991.
14. C. W. Ko and F. B. Shepherd. Bipartite domination and simultaneous matroid covers. *SIAM J. Discrete Math*, 16:517–523, 2003.
15. H. Lu and R. Ravi. The power of local optimization: Approximation algorithms for maximum-leaf spanning tree. In *Proceedings of the 30th Annual Allerton Conference on Communication, Control and Computing*, pages 533–542, 1992.
16. L. Trevisan. Non-approximability results for optimization problems on bounded degree instances. In *Proc. of the 33rd STOC*, 2001, pp. 453–461.
17. M. Zito. Maximum induced matching in regular graphs and trees. In *Proc. of the 25th WG*, LNCS 1665, pp. 89–100, 1999.

# Improved Online Algorithms for Buffer Management in QoS Switches

Marek Chrobak<sup>1</sup>, Wojciech Jawor<sup>1</sup>, Jiří Sgall<sup>2</sup>, and Tomáš Tichý<sup>2</sup>

<sup>1</sup> Department of Computer Science, University of California, Riverside, CA 92521.  
`marek,wojtek@cs.ucr.edu`

<sup>2</sup> Mathematical Institute, AS CR, Žitná 25, CZ-11567 Praha 1, Czech Republic.  
`sgall,tichy@math.cas.cz`

**Abstract.** The following buffer management problem is studied: packets with specified weights and deadlines arrive at a network switch and need to be forwarded so that the total weight of forwarded packets is maximized. A packet not forwarded before its deadline brings no profit. The main result is an online  $\frac{64}{33} \approx 1.939$ -competitive algorithm, the first deterministic algorithm for this problem with competitive ratio below 2. In the *s-uniform* case, where for all packets the deadline equals the arrival time plus *s*, we give an  $5 - \sqrt{10} \approx 1.838$ -competitive algorithm. This algorithm achieves the same ratio in a more general scenario when all packets are similarly ordered. For the 2-uniform case we give an algorithm with ratio  $\approx 1.377$  and a matching lower bound.

## 1 Introduction

One of the issues arising in IP-based QoS networks is how to manage the packet flows at a router level. In particular, in case of overloading, when the total incoming traffic exceeds the buffer size, the buffer management policy needs to determine which packets should be dropped by the router. Kesselman *et al.* [5] postulate that the packet drop policies can be modeled as combinatorial optimization problems. Our model is called *buffer management with bounded delay* in [5], and is defined as follows: Packets arrive at a network switch. Each packet is characterized by a positive weight and a deadline before which it must be transmitted. Packets can only be transmitted at integer time steps. If the deadline of a packet is reached while it is still being buffered, the packet is lost. The goal is to maximize the weighted number of forwarded packets.

This buffer management problem is equivalent to the online version of the following single-machine unit-job scheduling problem. We are given a set of unit-length jobs, with each job *j* specified by a triple  $(r_j, d_j, w_j)$ , where  $r_j$  and  $d_j$  are integral release times and deadlines, and  $w_j$  is a non-negative real weight. One job can be processed at each integer time. We use the term *weighted throughput* or *gain* for the total weight of the jobs completed by their deadline. The goal is to compute a schedule that maximizes the weighted throughput.

In the online version of the problem jobs arrive over time and the algorithm needs to schedule one of the pending jobs without knowledge of the future. An

online algorithm  $\mathcal{A}$  is  $R$ -competitive if its gain on any instance is at least  $1/R$  times the optimal gain on this instance. The *competitive ratio* of  $\mathcal{A}$  is the infimum of such values  $R$ . It is standard to view the online problem as a game between an online algorithm  $\mathcal{A}$  and an *adversary*, who issues the jobs and schedules them in order to maximize the ratio between his gain and the gain of  $\mathcal{A}$ .

**Past work.** A simple greedy algorithm that always schedules the heaviest available job is 2-competitive, and this is the best previous bound for deterministic algorithms for this problem. A lower bound of  $\phi \approx 1.618$  was shown in [1,3,4].

Some restrictions on instances of the problem have been studied in the literature [5,1,3]. Let the span of a job be the difference between its deadline and the release time. In  $s$ -uniform instances the span of each job is equal exactly  $s$ . In  $s$ -bounded instances, the span of each job is at most  $s$ . The lower bound of  $\phi \approx 1.618$  in [1,3,4] applies even to 2-bounded instances. A matching upper bound for the 2-bounded case was presented in [5]. Algorithms for 2-uniform instances were studied by Andelman *et al.* [1], who established a lower bound of  $\frac{1}{2}(\sqrt{3} + 1) \approx 1.366$  and an upper bound of  $\sqrt{2} \approx 1.414$ . This upper bound is tight for memoryless algorithms [2], that is, algorithms which base their decisions only on the weights of pending jobs and are invariant under scaling of weights. The first deterministic algorithms with competitive ratio lower than 2 for the  $s$ -bounded instances appear in [2].

Kesselman *et al.* [5,6] consider a different model related to the  $s$ -uniform case: packets do not have individual deadlines, instead they are stored in a buffer of capacity  $s$  and they are required to be served in FIFO order (i.e., if a packet is served, all packets in the buffer that arrived before the served one are dropped). Any algorithm in this FIFO model applies also to  $s$ -bounded model and has the same competitive ratio [6]. Recently, Bansal *et al.* [7] gave a deterministic 1.75-competitive algorithm in the FIFO model; this implies a 1.75-competitive algorithm for the  $s$ -uniform case.

A randomized 1.582-competitive algorithm for the general case was given in [2]. For 2-bounded instances, there is a 1.25-competitive algorithm [2] that matches the lower bound [3]. For the 2-uniform case the currently best lower bound for randomized algorithms is 1.172 [2].

**Our results.** Our main result is a deterministic  $64/33 \approx 1.939$ -competitive algorithm for the general case. This is the first deterministic algorithm for the problem with competitive ratio strictly below 2. All of the algorithms given previously in [5,1,2,6] achieved competitive ratio strictly below 2 only when additional restrictions on instances were placed.

For the  $s$ -uniform case, we give an algorithm with ratio  $5 - \sqrt{10} \approx 1.838$ . In fact, this result holds in a more general scenario when all jobs are similarly ordered, that is, when  $r_i < r_j$  implies  $d_i \leq d_j$  for all jobs  $i, j$ . Note that this includes  $s$ -uniform and 2-bounded instances, but not  $s$ -bounded ones for  $s \geq 3$ .

Finally, we completely solve the 2-uniform case: we give an algorithm with competitive ratio  $\approx 1.377$  and a matching lower bound. Note that this ratio is strictly in-between the previous lower and upper bounds from [1].

## 2 Terminology and Notation

A *schedule*  $S$  specifies which jobs are executed, and for each executed job  $j$  it specifies an integral time  $t$ ,  $r_j \leq t < d_j$ , when it is scheduled. Only one job can be scheduled at any  $t$ . The *throughput* or *gain* of a schedule  $S$  is the total weight of the jobs executed in  $S$ . If  $\mathcal{A}$  is a scheduling algorithm, by  $\text{gain}_{\mathcal{A}}(I)$  we denote the gain of the schedule computed by  $\mathcal{A}$  on  $I$ . The optimal gain on  $I$  is denoted by  $\text{opt}(I)$ . A job  $i$  is *pending* in  $S$  at time  $t$  if  $r_i \leq t < d_i$  and  $i$  has not been scheduled in  $S$  before  $t$ . (Thus all jobs released at time  $t$  are considered pending.) An instance is *s-bounded* if  $d_j - r_j \leq s$  for all jobs  $j$ . Similarly, an instance is *s-uniform* if  $d_j - r_j = s$  for all  $j$ . An instance is *similarly ordered* if  $r_i < r_j$  implies  $d_i \leq d_j$  for any two jobs  $i$  and  $j$ .

Given two jobs  $i, j$ , we say that  $i$  *dominates*  $j$  if either (i)  $d_i < d_j$ , or (ii)  $d_i = d_j$  and  $w_i > w_j$ , or (iii)  $d_i = d_j$ ,  $w_i = w_j$  and  $i < j$ . (Condition (iii) only ensures that ties are broken in some arbitrary but consistent way.) Given a non-empty set of jobs  $J$ , the *dominant* job in  $J$  is the one that dominates all other jobs in  $J$ ; it is always uniquely defined as ‘dominates’ is a linear order.

A schedule  $S$  is called *canonical earliest-deadline* if for any jobs  $i$  and  $j$  in  $S$ , where  $i$  is scheduled at time  $t$  and  $j$  scheduled later, either  $j$  is released strictly after time  $t$ , or  $i$  dominates  $j$ . In other words, at any time, the job to be scheduled dominates all pending jobs that appear later in  $S$ . Any schedule can be easily converted into a canonical earliest-deadline schedule by rearranging its jobs. Thus we may assume that offline schedules are canonical earliest-deadline.

## 3 A 64/33-Competitive Algorithm

We start with some intuitions that should be helpful in understanding the algorithm and its analysis. The greedy algorithm that always executes the heaviest job (H-job) is not better than 2-competitive. An alternative idea is to execute the earliest deadline job at each step. This algorithm is not competitive at all, as it could execute many small weight jobs even if there are heavy jobs pending with only slightly larger deadlines. A natural refinement of this approach is to focus on sufficiently heavy jobs, of weight at least  $\alpha$  times the maximal weight, and chose the earliest deadline job among those (an E-job). As it turns out, this algorithm is also not better than 2-competitive.

The general idea of our new algorithm is to alternate H-jobs and E-jobs. Although this simple algorithm, as stated, still has ratio no better than 2, we can reduce the ratio by introducing some minor modifications.

**Algorithm GENFLAG:** We use parameters  $\alpha = \frac{7}{11}$ ,  $\beta = \frac{8}{11}$ , and a Boolean variable  $E$ , initially set to 0, that stores information about the previous step. At a given time step  $t$ , update the set of pending jobs (remove jobs with deadline  $t$  and add jobs released at  $t$ ). If there are no pending jobs, go to the next time step. Otherwise, let  $h$  be the heaviest pending job (breaking ties in favor of dominant jobs) and  $e$  the dominant job among the pending jobs with weight at least  $\alpha w_h$ . Schedule either  $e$  or  $h$  according to the following procedure:



```

if  $E = 0$ 
  then schedule  $e$ ; if  $e \neq h$  then set  $E \leftarrow 1$ 
  else if  $d_e = t + 1$  and  $w_e \geq \beta w_h$  then schedule  $e$  else schedule  $h$ ;
  set  $E \leftarrow 0$ 

```

A job  $e$  scheduled while  $E = 0$  is called an *O-job* if  $e = h$ , or an *E-job* otherwise. A job  $e$  scheduled while  $E = 1$  is an *U-job*. A job  $h$  scheduled in the last case is an *H-job*. The letters stand for **O**bvious, **E**arly, **U**rgent, and **H**eaviest.

Variable  $E$  is 1 iff the previous job was an E-job. Thus, in terms of the labels, the algorithm proceeds as follows: If an O-job is available, we execute it. Otherwise, we execute an E-job, and in the next step either a U-job (if available) or an H-job. (There always is a pending job at this next step, thanks to the condition  $e \neq h$ : if  $d_h = t$  then  $e = h$  by the definition of dominance; so if an E-job is executed at time  $t$ ,  $d_h > t$  and  $h$  is pending in the next step.)

**Theorem 3.1.** *GENFLAG is a  $64/33$ -competitive deterministic algorithm for unit-job scheduling.*

*Proof.* The proof is by a charging scheme. Fix an arbitrary (offline) schedule ADV. For each job  $j$  executed in ADV, we partition its weight  $w_j$  into one or two *charges*, and assign each charge to a job executed by GENFLAG. If the total charge to each job  $i$  of GENFLAG were at most  $Rw_i$ , the  $R$ -competitiveness of GENFLAG would follow by summation over all jobs. Our charging scheme does not always meet this simple condition. Instead, we divide the jobs of GENFLAG into disjoint groups, where each group is either a single O-job, or an EH-pair (an E-job followed by an H-job), or an EU-pair (an E-job followed by a U-job). This is possible by the discussion of types of jobs before the theorem. For each group we prove that its *charging ratio* is at most  $R$ , where the charging ratio is defined as the total charge to this group divided by the total weight of the group. This implies that GENFLAG is  $R$ -competitive by summation over all groups.

**Charging scheme.** Let  $j$  be the job executed at time  $t$  in ADV. Denote by  $i$  and  $h$ , respectively, the job executed by GENFLAG and the heaviest pending job at time  $t$ . (If there are no pending jobs, introduce a “dummy” job of weight 0. This does not change the algorithm.) Then  $j$  is charged to GENFLAG’s jobs, according to the following rules.

- (EB) If  $j$  is executed by GENFLAG before time  $t$ , then charge  $(1 - \beta)w_h$  to  $i$  and the remaining  $w_j - (1 - \beta)w_h$  to  $j$ .
- (EF) Else, if  $j$  is executed by GENFLAG after time  $t$ , then charge  $\beta w_j$  to  $i$  and  $(1 - \beta)w_j$  to  $j$ .
- (NF) Else, if  $i$  is an H-job,  $w_j \geq \beta w_i$ , and ADV executes  $i$  after time  $t$ , then charge  $\beta w_j$  to  $i$  and  $(1 - \beta)w_j$  to the job executed by GENFLAG at time  $t + 1$ .
- (U) Else, charge  $w_j$  to  $i$ . (Note that this case includes the case  $i = j$ .)

We label all charges as EB, EF, NF, U, according to which case above applies. We also distinguish upward, forward, and backward charges, defined in the obvious way. Thus, for example, in case (EB), the charge of  $w_j - (1 - \beta)w_h$  to  $j$



is a backward EB-charge. The letters in the labels refer to whether  $j$  was executed by GENFLAG, and to the charge direction: **Executed-Backward**, **Executed-Forward**, **Non-executed-Forward**, **Upward**. We start with several simple observations used later in the proof, sometimes without explicit reference.

**Observation 1:** At time  $t$ , let  $h$  be the heaviest job of GENFLAG and  $e$  the dominant job of weight at least  $\alpha w_h$ . Let  $j$  be any pending job. Then

$$(1.1) \quad w_j \leq w_h.$$

$$(1.2) \quad \text{If } j \text{ dominates } e \text{ then } w_j < \alpha w_h.$$

*Proof:* Inequality (1.1) is trivial, by the definition of  $h$ . Inequality (1.2) follows from the fact that  $e$  dominates all pending jobs with weight at least  $\alpha w_h$ .

**Observation 2:** Suppose that at time  $t$  GENFLAG schedules a job that receives a forward NF-charge from the job  $l$  scheduled at time  $t - 1$  in ADV. Then  $l$  is pending at time  $t$ .

*Proof:* Denote by  $f$  the heaviest pending job of GENFLAG at time  $t - 1$ . By the definition of NF-charges,  $l \neq f$ ,  $l$  is pending at time  $t - 1$ ,  $f$  is executed as an H-job,  $d_f \geq t + 1$ , and  $w_l \geq \beta w_f$ . Therefore  $d_l \geq t + 1$ , since otherwise GENFLAG would execute  $l$  (or some other job) as a U-job at step  $t - 1$ . Thus  $l$  is also pending at step  $t$ , as claimed.

**Observation 3:** Suppose that at time  $t$  GENFLAG schedules a job  $e$  of type O or E, and  $h$  is the heaviest pending job at time  $t$ . Then

$$(3.1) \quad \text{The upward charge to } e \text{ is at most } w_h.$$

$$(3.2) \quad \text{If ADV executes } e \text{ after time } t \text{ then the upward charge is at most } \alpha w_h.$$

$$(3.3) \quad \text{The forward NF-charge to } e \text{ is at most } (1 - \beta)w_h.$$

*Proof:* Denote by  $j$  the job executed at time  $t$  in ADV. If  $j$  is scheduled by GENFLAG before time  $t$ , then the upward charge is  $(1 - \beta)w_h \leq \alpha w_h$  and claims (3.1) and (3.2) hold. Otherwise  $j$  is pending at time  $t$ . Then the upward charge is at most  $w_j \leq w_h$  and (3.1) follows. If ADV executes  $e$  after time  $t$ , then, since ADV is canonical, job  $j$  must dominate  $e$ , and (3.2) follows from (1.2). (3.3) follows by Observation 2 and (1.1).

**Observation 4:** Suppose that at time  $t$  GENFLAG executes an H-job  $h$  that is executed after time  $t$  in ADV. Then the upward charge to  $h$  is at most  $\beta w_h$ .

*Proof:* Let  $j$  be the job executed at time  $t$  in ADV. In case (EB), the upward charge to  $h$  is at most  $(1 - \beta)w_h \leq \beta w_h$ . In all other cases,  $j$  is pending at time  $t$ , so  $w_j \leq w_h$ . In cases (EF) and (NF), the charge is  $\beta w_j \leq \beta w_h$ . In case (U) the charge is  $w_j$ , but since (NF) did not apply, this case can occur only if  $w_j \leq \beta w_h$ .

Now we examine the charges to all job groups in GENFLAG's schedule.

**O-jobs.** Let  $e = h$  be an O-job executed at time  $t$ . The forward NF-charge is at most  $(1 - \beta)w_e$ . If  $e$  gets a backward EB-charge then  $e$  gets no forward EF-charge and the upward charge is at most  $\alpha w_e$ ; the total charging ratio is at most  $2 + \alpha - \beta < R$ . If  $e$  does not get a backward EB-charge then the forward EF-charge is at most  $(1 - \beta)w_e$  and the upward charge is at most  $w_e$ ; the charging ratio is at most  $3 - 2\beta < R$ .

**E-jobs.** Before considering EH-pairs and EU-pairs, we estimate separately the charges to E-jobs. Suppose that at time  $t$  GENFLAG executes an E-job  $e$ , and

the heaviest job is  $h$ . We claim that  $e$  is charged at most  $\alpha w_h + (2 - \beta)w_e$ . We have two cases.

**Case 1:**  $e$  gets no backward EB-charge. Then, in the worst case,  $e$  gets an upward charge of  $w_h$ , a forward NF-charge of  $(1 - \beta)w_h$ , and a forward EF-charge of  $(1 - \beta)w_e$ . Using  $(2 - \beta)w_h = 2\alpha w_h$  and  $\alpha w_h \leq w_e$ , the total charge is at most  $(2 - \beta)w_h + (1 - \beta)w_e \leq \alpha w_h + (2 - \beta)w_e$  as claimed.

**Case 2:**  $e$  gets a backward EB-charge. Then there is no forward EF-charge and the upward charge is at most  $\alpha w_h$  by (3.2). Let  $l$  be the job scheduled at time  $t - 1$  in ADV. If there is an NF-charge, it is generated by  $l$  and then  $l$  is pending for GENFLAG at time  $t$  by Observation 2.

If there is a forward NF-charge and  $d_l \geq d_e$ , then  $l$  is pending at the time when ADV schedules  $e$ . (In this case  $l$  cannot be executed by GENFLAG, because charges EB, EF did not apply to  $l$ .) Consequently,  $e$  receives at most a backward EB-charge  $w_e - (1 - \beta)w_l$ , a forward NF-charge  $(1 - \beta)w_l$ , and an upward charge  $\alpha w_h$ . The total is at most  $\alpha w_h + w_e \leq \alpha w_h + (2 - \beta)w_e$ , as claimed.

Otherwise, there is no forward NF-charge, or there is a forward NF-charge and  $d_l < d_e$ . In the second case,  $l$  is pending at  $t$ , and  $l$  dominates  $e$ , so the forward NF-charge is at most  $(1 - \beta)w_l \leq (1 - \beta)w_e$ . With the backward EB-charge of  $w_e$  and upward charge of at most  $\alpha w_h$ , the total is at most  $\alpha w_h + (2 - \beta)w_e$ , as claimed.

**EH-pairs.** Let  $e$  be the E-job scheduled at time  $t$ ,  $h$  the heaviest pending job at time  $t$ , and  $h'$  the H-job at time  $t + 1$ . By the algorithm,  $e \neq h$ . Note that, since GENFLAG did not execute  $h$  as an O-job at time  $t$ ,  $h$  is still pending after the E-step and  $w_{h'} \geq w_h$ .

We now estimate the charge to  $h'$ . There is no forward NF-charge, as the previous step is not an H-step. If there is a backward EB-charge, the additional upward charge is at most  $\beta w_{h'}$  by Observation 4 and the total is at most  $(1 + \beta)w_{h'}$ . If there is no EB-charge, the sum of the upward charge and a forward EF-charge is at most  $w_{h'} + (1 - \beta)w_{h'} \leq (1 + \beta)w_{h'}$ . With the charge to  $e$  of at most  $\alpha w_h + (2 - \beta)w_e$ , the total charge of the EH-pair is at most  $\alpha w_h + (2 - \beta)w_e + (1 + \beta)w_{h'}$ , and thus the charging ratio is at most

$$2 - \beta + \frac{\alpha w_h + (2\beta - 1)w_{h'}}{w_e + w_{h'}} \leq 2 - \beta + \frac{\alpha w_h + (2\beta - 1)w_{h'}}{\alpha w_h + w_{h'}} = R.$$

The first step follows from  $w_e \geq \alpha w_h$ . As  $2\beta - 1 < 1$ , the next expression is decreasing in  $w_{h'}$ , so the maximum is at  $w_{h'} = w_h$ , and it is equal to  $R$  by the definitions of  $\alpha$  and  $\beta$ .

**EU-pairs.** As in the previous case, let  $e$ , and  $h$  denote the E-job scheduled at time  $t$  and the heaviest pending job at time  $t$ . By  $g$  and  $h'$  we denote the scheduled U-job and the heaviest pending job at time  $t + 1$ . As in the case of EH-pairs,  $e \neq h$  and  $w_{h'} \geq w_h$ .

Job  $g$  gets no backward EB-charge, since it expires, and no forward NF-charge, since the previous step is not an H-step. The upward charge is at most  $w_{h'}$ , the forward EF-charge is at most  $(1 - \beta)w_g$ .

With the charge to  $e$  of at most  $\alpha w_h + (2 - \beta)w_e$ , the charge of the EU-pair is at most  $\alpha w_h + (2 - \beta)w_e + w_{h'} + (1 - \beta)w_g$ , so the charging ratio is at most

$$2 - \beta + \frac{\alpha w_h + w_{h'} - w_g}{w_e + w_g} \leq 2 - \beta + \frac{\alpha w_h + (1 - \beta)w_{h'}}{\alpha w_h + \beta w_{h'}} \leq 2 - \beta + \frac{\alpha + 1 - \beta}{\alpha + \beta} = R.$$

In the first step, we apply bounds  $w_e \geq \alpha w_h$  and  $w_g \geq \beta w_{h'}$ . As  $1 - \beta < \beta$ , the next expression is decreasing in  $w_{h'}$ , so the maximum is at  $w_{h'} = w_h$ .  $\square$

## 4 Similarly Ordered Jobs

We now consider the case when the jobs are similarly ordered, that is  $r_i < r_j$  implies  $d_i \leq d_j$  for all jobs  $i, j$ .

**Algorithm SIMFLAG:** We use one parameter  $\alpha = \sqrt{10}/5 \approx 0.633$  and a Boolean variable  $E$ , initially set to 0. At a given time step  $t$ , update the set of pending jobs. If there are no pending jobs, go to the next time step. Otherwise, let  $h$  be the heaviest pending job (breaking ties in favor of dominant jobs) and  $e$  the dominant job among the pending jobs with weight at least  $\alpha w_h$ . Schedule either  $e$  or  $h$  according to the following procedure:

**if**  $E = 0$  **then** schedule  $e$ ; **if**  $e \neq h \wedge d_e > t + 1$  **then** set  $E \leftarrow 1$   
**else** schedule  $h$ ; set  $E \leftarrow 0$

A job  $e$  scheduled while  $E = 0$  is called an *O-job* if  $e = h$  or  $d_e = t + 1$ , and an *E-job* otherwise. A job  $h$  scheduled while  $E = 1$  is called an *H-job*. The intuition behind the algorithm is similar to GENFLAG, and, in fact, it is simpler. It schedules O-jobs, if available, and if not, it schedules an E-job followed by an H-job. (By the condition  $e \neq h$ , there is a pending job in the next step.)

**Theorem 4.1.** *SIMFLAG is a  $5 - \sqrt{10} \approx 1.838$ -competitive deterministic algorithm for unit-job scheduling for similarly ordered instances.*

The proof is by a charging scheme, similarly as for GENFLAG. The key observation is that if after scheduling an *E-job* at time  $t$ , there is a pending job with deadline  $t + 2$  or more, no job with deadline  $t + 1$  can be released, and thus we do not need the special case of *U-jobs*. This simplifies both the algorithm and analysis, and improves the competitive ratio.

We describe the charging scheme below. The rest of the proof showing that the charging ratio to each group is at most  $R$  is similar as for GENFLAG and is omitted in this extended abstract.

**Charging scheme.** Let  $j$  be the job executed at time  $t$  in ADV. Denote by  $i$  and  $h$ , respectively, the job executed by SIMFLAG and the heaviest pending job at time  $t$ . (Without loss of generality, we can assume that such jobs exist.) Let  $\beta = 4 - \sqrt{10} \approx 0.838$ . Then  $j$  is charged to SIMFLAG's jobs, according to the following rules.

- (EB) If  $j$  is executed by SIMFLAG at or before time  $t$ , then charge  $w_j$  to  $j$ .
- (NF) Else, if  $d_j > t + 1$  and  $i$  is an H-job, then charge  $\beta w_j$  to  $i$  and  $(1 - \beta)w_j$  to the job scheduled by SIMFLAG at time  $t + 1$ .
- (U) Else, charge  $w_j$  to  $i$ .

## 5 2-Uniform Instances

Let  $Q \approx 1.377$  be the largest root of  $Q^3 + Q^2 - 4Q + 1 = 0$ . First we prove a lower bound of  $Q$  on the competitive ratio and then we give a matching algorithm, both for the 2-uniform instances, i.e., each job  $j$  satisfies  $d_j = r_j + 2$ .

At each step  $t$ , we distinguish *old pending jobs*, that is, those that were released at time  $t - 1$  but not executed, from the newly released jobs. We can always ignore all the old pending jobs except for the heaviest one, as only one of the old pending jobs can be executed. To simplify notation, we identify jobs by their weight. Thus “job  $x$ ” means the job with weight  $x$ . Such a job is usually uniquely defined by the context, possibly after specifying if it is an old pending job or a newly released job.

### 5.1 Lower Bound

Fix some  $0 < \epsilon < 2Q - 2$ . We define a sequence  $\Psi_i$ ,  $i = 1, 2, \dots$ , as follows. For  $i = 1$ ,  $\Psi_1 = Q - 1 - \epsilon$ . Inductively, for  $i \geq 1$ , let

$$\Psi_{i+1} = \frac{(2 - Q)\Psi_i - (Q - 1)^2}{2 - Q - \Psi_i}.$$

**Lemma 5.1.** *For all  $i$ , we have  $|\Psi_i| < Q - 1 \approx 0.377$ . Furthermore, the sequence  $\{\Psi_i\}$  converges to  $1 - Q \approx -0.377$ .*

*Proof.* Substituting  $z_i = \Psi_i + Q - 1$ , we get  $z_{i+1} = \frac{(3-2Q)z_i}{1-z_i}$ . We show inductively  $0 < z_i \leq 2Q - 2 - \epsilon$ . Initially,  $z_1 = 2Q - 2 - \epsilon$ . Inductively, if  $0 < z_i \leq 2Q - 2 - \epsilon$ , then  $0 < z_{i+1} \leq \frac{3-2Q}{3-2Q+\epsilon} z_i < z_i$ . Thus  $0 < z_i \leq 2Q - 2 - \epsilon$  for all  $i$ ,  $\lim_{i \rightarrow \infty} z_i = 0$ , and the lemma follows.  $\square$

**Theorem 5.2.** *There is no deterministic online algorithm for the 2-uniform case with competitive ratio smaller than  $Q$ .*

*Proof.* The proof is by contradiction. Suppose that  $\mathcal{A}$  is a  $(Q - \epsilon)$ -competitive algorithm, for some  $\epsilon > 0$ . We develop an adversary strategy that forces  $\mathcal{A}$ 's ratio to be bigger than  $Q - \epsilon$ .

Let  $\Psi_i$  be as defined before Lemma 5.1. For  $i \geq 1$  define

$$a_i = \frac{1 - \Psi_i}{Q - 1} \quad \text{and} \quad b_i = \frac{Q(2 - Q - \Psi_i)}{(Q - 1)^2}.$$

By Lemma 5.1, for all  $i$ ,  $b_i > a_i > 1$  and, for large  $i$ ,  $a_i \approx 3.653$  and  $b_i \approx 9.688$ .

Our strategy proceeds in iterations. It guarantees that at the beginning of iteration  $i$ , both  $\mathcal{A}$  and the adversary have one or two old pending job(s) of the same weight  $x_i$ . Note that it is irrelevant if one or two old pending jobs are present, and also if they are the same for  $\mathcal{A}$  and the adversary.

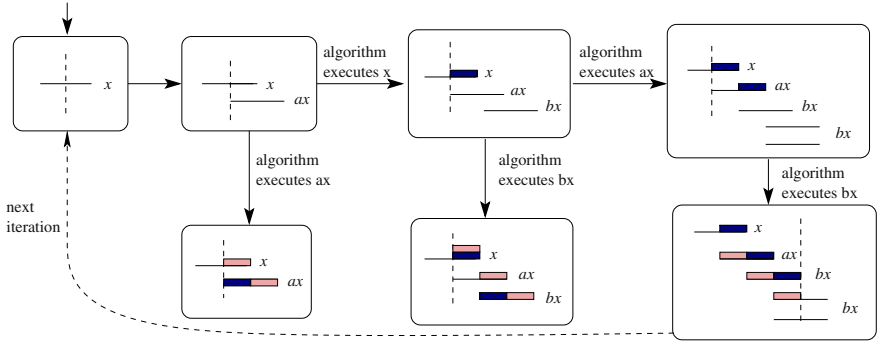
Initially, we issue two jobs of some arbitrary weight  $x_1 > 0$  at time 0 and start iteration 1 at time 1. Both  $\mathcal{A}$  and the adversary execute one job  $x_1$  and both have an old pending job with weight  $x_1$  at the beginning of iteration 1.

In iteration  $i$ ,  $\mathcal{A}$  and the adversary start with an old pending job  $x_i$ . The adversary now follows this procedure:

- issue one job  $a_i x_i$
- (A) if  $\mathcal{A}$  executes  $a_i x_i$  then execute  $x_i, a_i x_i$  and halt  
 else ( $\mathcal{A}$  executes  $x_i$ )  
 at the next time step issue  $b_i x_i$
- (B) if  $\mathcal{A}$  executes  $b_i x_i$  then execute  $x_i, a_i x_i, b_i x_i$ , and halt  
 else ( $\mathcal{A}$  executes  $a_i x_i$ )
- (C) at the next time step issue two jobs  $x_{i+1} = b_i x_i$   
 execute  $a_i x_i, b_i x_i, b_i x_i$

If  $\mathcal{A}$  executes first  $x_i$  and then  $a_i x_i$ , then after step (C) it executes one job  $b_i x_i$ , either the old pending one or one of the two newly released jobs. After this, both  $\mathcal{A}$  and the adversary have one or two newly released jobs  $b_i x_i$  pending, and the new iteration starts with  $x_{i+1} = b_i x_i$ .

A single complete iteration of the adversary strategy is illustrated in Figure 1.



**Fig. 1.** The adversary strategy. We denote  $x = x_i$ ,  $a = a_i$ , and  $b = b_i$ . Line segments represent jobs, dark rectangles slots when the job is executed by  $\mathcal{A}$ , and lightly shaded rectangles executions by the adversary.

If the game completes  $i - 1$  iterations, then define  $gain_i$  and  $adv_i$  to be the gain of  $\mathcal{A}$  and the adversary in these steps. We claim that for any  $i$ , either the adversary wins the game before iterations  $i$ , or else in iteration  $i$  we have

$$(Q - \epsilon)gain_i - adv_i \leq \Psi_i x_i. \quad (1)$$

The proof is by induction. For  $i = 1$ ,  $(Q - \epsilon)gain_1 - adv_1 \leq (Q - \epsilon - 1)x_1 = \Psi_1 x_1$ .

Suppose that (1) holds after iteration  $i - 1$ , that is, when  $\mathcal{A}$  and the adversary have an old pending job with weight  $x_i$ . If  $\mathcal{A}$  executes  $a_i x_i$  in step (A), then, denoting by  $\rho$  the sequence of all released jobs (up to and including  $a_i x_i$ ), using the inductive assumption, and substituting the formula for  $a_i$ , we have

$$\begin{aligned} (Q - \epsilon)gain_{\mathcal{A}}(\rho) - adv(\rho) &= (Q - \epsilon)gain_i - adv_i + (Q - \epsilon)a_i x_i - (x_i + a_i x_i) \\ &\leq [\Psi_i - 1 + (Q - \epsilon - 1)a_i]x_i = -\epsilon a_i x_i < 0, \end{aligned}$$

contradicting the  $(Q - \epsilon)$ -competitiveness of  $\mathcal{A}$ .

If  $\mathcal{A}$  executes  $x_i$  and then  $b_i x_i$  in (B), then, again, denoting by  $\rho$  the sequence of all released jobs, using the inductive assumption, and substituting the formulas for  $a_i, b_i$ , we obtain a contradiction with the  $(Q - \epsilon)$ -competitiveness of  $\mathcal{A}$ .

$$\begin{aligned} (Q - \epsilon)gain_{\mathcal{A}}(\rho) - adv(\rho) &= (Q - \epsilon)gain_i - adv_i + (Q - \epsilon)(x_i + b_i x_i) - (x_i + a_i x_i + b_i x_i) \\ &\leq [\Psi_i + Q - \epsilon - 1 + (Q - \epsilon - 1)b_i - a_i]x_i = -\epsilon(1 + b_i)x_i < 0. \end{aligned}$$

In the remaining possibility (C),  $\mathcal{A}$  executes first  $x_i$ , then  $a_i x_i$ , and then  $b_i x_i$ . Using the formulas for  $a_i, b_i, \Psi_{i+1}$ , and the defining equation  $Q^3 + Q^2 - 4Q + 1 = 0$ , we have

$$\begin{aligned} (Q - \epsilon)gain_{i+1} - adv_{i+1} &\leq (Q - \epsilon)gain_i - adv_i + (Q - \epsilon)(x_i + a_i x_i + b_i x_i) - (a_i x_i + 2b_i x_i) \\ &\leq [\Psi_i + Q + (Q - 1)a_i - (2 - Q)b_i]x_i = b_i \Psi_{i+1} x_i = x_{i+1} \Psi_{i+1}. \end{aligned}$$

This completes the inductive proof of (1).

Lemma 5.1 and (1) imply that, for  $i$  large enough, we have  $(Q - \epsilon)gain_i - adv_i \leq \Psi_i x_i < (1 - Q + \epsilon)x_i$ . Denoting by  $\rho$  the sequence of all jobs (including the pending jobs  $x_i$ ), we have

$$\begin{aligned} (Q - \epsilon)gain_{\mathcal{A}}(\rho) - adv(\rho) &= (Q - \epsilon)gain_i - adv_i + (Q - \epsilon)x_i - x_i \\ &< (1 - Q + \epsilon)x_i + (Q - \epsilon)x_i - x_i = 0. \end{aligned}$$

This contradicts the  $(Q - \epsilon)$ -competitiveness of  $\mathcal{A}$ .  $\square$

## 5.2 Upper Bound

We now present our  $Q$ -competitive algorithm for 2-uniform case. Given that the 2-uniform case seems to be the most elementary case of unit job scheduling (without being trivial), our algorithm (and its analysis) is surprisingly difficult. Recall, however, that, as shown in [2], any algorithm for this case with competitive ratio below  $\sqrt{2}$  needs to use some information about the past. Further, when the adversary uses the strategy from Theorem 5.2, any  $Q$ -competitive algorithm needs to behave in an essentially unique way. Our algorithm was designed to match this optimal strategy, and then extended (by interpolation) to other adversarial strategies. Thus we suspect that the complexity of the algorithm is inherent in the problem and cannot be avoided.

We start with some intuitions. Let  $\mathcal{A}$  be our online algorithm. Suppose that at time  $t$  we have one old pending job  $z$ , and two new pending jobs  $b, c$  with  $b \geq c$ . In some cases, the decision which job to execute is easy. If  $c \geq z$ ,  $\mathcal{A}$  can ignore  $z$  and execute  $b$  in the current step. If  $z \geq b$ ,  $\mathcal{A}$  can ignore  $c$  and execute  $z$  in the current step. If  $c < z < b$ ,  $\mathcal{A}$  faces a dilemma: it needs to decide whether to execute  $z$  or  $b$ . For  $c = 0$ , the choice is based on the ratio  $z/b$ . If  $z/b$  exceeds a certain threshold (possibly dependent on the past), we execute  $z$ , otherwise we execute  $b$ . Taking those constraints into account, and interpolating for arbitrary

values of  $c$ , we can handle all cases by introducing a parameter  $\eta$ ,  $0 \leq \eta \leq 1$ , and making the decision according to the following procedure:

**Procedure  $\text{SSP}_\eta$ :** If  $z \geq \eta b + (1 - \eta)c$  schedule  $z$ , otherwise schedule  $b$ .

To derive an online algorithm, say  $\mathcal{A}$ , we examine the adversary strategy in the lower bound proof. Consider the limit case, when  $i \rightarrow \infty$ , and let  $a_* = \lim_{i \rightarrow \infty} a_i = Q/(Q - 1)$  and  $b_* = \lim_{i \rightarrow \infty} b_i = Q/(Q - 1)^2$ . Assume that in the previous step two jobs  $z$  were issued. If the adversary now issues a single job  $a$ , then  $\mathcal{A}$  needs to do the following: if  $a < a_*z$ , execute  $z$ , and if  $a \geq a_*z$ , then execute  $a$ . Thus in this case we need to apply  $\text{SSP}_\alpha$  with the threshold  $\alpha = 1/a_* = (Q - 1)/Q$ .

Then, suppose that in the first step  $\mathcal{A}$  executed  $z$ , so that in the next step  $a$  is pending. If the adversary now issues a single job  $b$ , then (assuming  $a \approx a_*$ )  $\mathcal{A}$  must to do the following: if  $b \leq b_*z$ , execute  $a$ , and if  $b \geq b_*z$ , then execute  $b$ . Thus in this case we need to apply  $\text{SSP}_\beta$  with the threshold  $\beta = a_*/b_* = Q - 1$ .

Suppose that we execute  $a$ . In the lower-bound strategy, the adversary would now issue two jobs  $b$  in the next step. But what happens if he issues a single job, say  $c$ ? Calculations show that  $\mathcal{A}$ , in order to be  $Q$ -competitive, needs to use now yet a different parameter  $\eta$  in  $\text{SSP}_\eta$ . This parameter is not uniquely determined, but it must be at least  $\gamma = (3 - 2Q)/(2 - Q) > Q - 1$ . Further, it turns out that the same value  $\gamma$  can be used on subsequent single-job requests.

Our algorithm is derived from the above analysis: on a sequence of single-job requests in a row, use  $\text{SSP}_\eta$  with parameter  $\alpha$  in the first step, then  $\beta$  in the second step, and  $\gamma$  in all subsequent steps. In general, of course, two jobs can be issued at each step (or more, but only two heaviest jobs need to be considered.) We think of an algorithm as a function of several arguments. The values of this function on the boundary are determined from the optimal adversary strategy, as explained above. The remaining values are obtained through interpolation.

We now give a formal description of our algorithm. Let

$$\alpha = \frac{Q - 1}{Q} \approx 0.27, \quad \beta = Q - 1 \approx 0.38, \quad \gamma = \frac{3 - 2Q}{2 - Q} \approx 0.39,$$

$$\lambda(\xi) = \min \left\{ 1, \frac{\xi - \alpha}{\beta - \alpha} \right\}, \quad \delta(\mu, \xi) = \mu\alpha + (1 - \mu)[\beta + (\gamma - \beta)\lambda(\xi)],$$

where  $0 \leq \mu \leq 1$  and  $\alpha \leq \xi \leq \gamma$ . Note that for the parameters  $\mu, \xi$  within their ranges, we have  $0 \leq \lambda(\xi) \leq 1$ ,  $\alpha \leq \delta(\mu, \xi) \leq \gamma$ . Function  $\delta$  also satisfies  $\delta(1, \xi) = \alpha$ ,  $\delta(0, \xi) \geq \beta$  for any  $\xi$ , and  $\delta(0, \alpha) = \beta$ ,  $\delta(0, \beta) = \gamma$ .

**Algorithm SWITCH.** Fix a time step  $t$ . Let  $u, v$  (where  $u \geq v$ ) be the two jobs released at time  $t - 1$ , and  $b, c$  (where  $b \geq c$ ) be the jobs released at time  $t$ . Let  $z \in \{u, v\}$  be the old pending job. Let also  $\xi$  be the parameter of  $\text{SSP}$  used at time  $t - 1$  (initially  $\xi = \alpha$ ). Then choose the job to execute as follows:

- If  $z = u$  run  $\text{SSP}_\eta$  with  $\eta = \delta(v/u, \xi)$ ,
- If  $z = v$  run  $\text{SSP}_\eta$  with  $\eta = \alpha$ .

**Theorem 5.3.** *Algorithm SWITCH is  $Q$ -competitive for the 2-uniform case, where  $Q \approx 1.377$  is the largest root of  $Q^3 + Q^2 - 4Q + 1 = 0$ .*

The proof of the theorem is by tedious case analysis of an appropriate potential function and will appear in the full version of this paper.

## Conclusions

We established the first upper bound better than 2 on the competitiveness of deterministic scheduling unit jobs to maximize weighted throughput. There is still a wide gap between our upper bound of  $\approx 1.939$  and the best known lower bound of  $\phi$ . Closing or substantially reducing this gap is a challenging open problem. We point out that our algorithm GENFLAG is not memoryless, as it uses one bit of information about the previous step. Whether it is possible to reduce the ratio of 2 with a memoryless algorithm remains an open problem.

Another open problem is to determine the competitiveness for similarly ordered or  $s$ -uniform instances. The best lower bound for similarly ordered instances is  $\phi$  (from the 2-bounded case), our algorithm is  $\approx 1.838$ -competitive. Our algorithm does not obey the FIFO rule, on the other hand, the 1.75-competitive algorithm from the FIFO model [7] does not apply to similarly ordered instances. It would be interesting to obtain any algorithm that obeys the FIFO rule and is better than 2-competitive for similarly ordered instances.

**Acknowledgments.** Chrobak and Jawor supported by NSF grants CCR-9988360 and CCR-0208856. Sgall and Tichý supported by Institute for Theoretical Computer Science, Prague (project LN00A056 of MŠMT ČR) and grant IAA1019401 of GA AV ČR.

## References

1. N. Andelman, Y. Mansour, and A. Zhu. Competitive queueing policies in QoS switches. In *Proc. 14th SODA*, pp. 761–770. ACM/SIAM, 2003.
2. Y. Bartal, F. Y. L. Chin, M. Chrobak, S. P. Y. Fung, W. Jawor, R. Lavi, J. Sgall, and T. Tichý. Online competitive algorithms for maximizing weighted throughput of unit jobs. In *Proc. 21st STACS*, LNCS 2996, pp. 187–198. Springer, 2004.
3. F. Y. L. Chin and S. P. Y. Fung. Online scheduling for partial job values: Does timesharing or randomization help? *Algorithmica*, 37:149–164, 2003.
4. B. Hajek. On the competitiveness of online scheduling of unit-length packets with hard deadlines in slotted time. In *Conf. in Information Sciences and Systems*, pp. 434–438, 2001.
5. A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in QoS switches. In *Proc. 33rd STOC*, pp. 520–529. ACM, 2001.
6. A. Kesselman, Y. Mansour, and R. van Stee. Improved competitive guarantees for QoS buffering. In *Proc. 11th ESA*, LNCS 2832, pp. 361–372. Springer, 2003.
7. M. Mahdian, N. Bansal, L. Fleischer, T. Kimbrel, B. Schieber, and M. Sviridenko. Further improvements in competitive guarantees for QoS buffering. In *Proc. 31st ICALP*. Springer, 2004.



# Time Dependent Multi Scheduling of Multicast<sup>\*</sup>

Rami Cohen<sup>1</sup>, Dror Rawitz<sup>2</sup>, and Danny Raz<sup>1</sup>

<sup>1</sup> Department of Computer Science, Technion, Haifa 32000, Israel

{`ramic,danny`}@cs.technion.ac.il

<sup>2</sup> Department of Electrical Engineering, Tel-Aviv University, Tel-Aviv 69978, Israel

rawitz@eng.tau.ac.il

**Abstract.** Many network applications that need to distribute contents and data to a large number of clients use a hybrid scheme in which one or more multicast channel is used in parallel to a unicast dissemination. This way the application can distribute data using one of its available multicast channels or by sending one or more unicast transmissions. In such a model the utilization of the multicast channels is critical for the overall performance of the system. We study the scheduling algorithm of the sender in such a model. We describe this scheduling problem as an optimization problem where the objective is to maximize the utilization of the multicast channel. Our model captures the fact that it may be beneficial to multicast an object more than once (e.g. page update). Thus, the benefit depends, among other things, on the last time the object was sent, which makes the problem much more complex than previous related scheduling problems. Using the *local ratio technique* we obtain a 4-approximation algorithm for the case where the objects are of fixed size and a 10-approximation algorithm for the general case. We also consider a special case which may be of practical interest, and prove that a simple greedy algorithm is a 3-approximation algorithm in this case.

## 1 Introduction

Web caching is a common way to overcome problems such as congestion and delay that often arise in the current Internet. Using this mechanism, web caches are deployed between web clients and web servers and store a subset of the server content. Upon receiving a request from a client (or from another cache) a web cache tries to satisfy this request using a local (valid) copy of the required object. If the cache does not hold such a copy, it tries to retrieve the object from other caches (e.g. using Internet Cache Protocol [15]), or from the original server (e.g. using HTTP [7]).

Web caching can be useful in several aspects. From the client's point of view, the presence of caches reduces the average response time. Internet Service Providers (ISP's) deploy web caches in order to reduce the load over their links to the rest of the Internet. Web servers benefit from web cache deployment as

---

<sup>\*</sup> This work was partly supported by the CONTEXT IST-2001-38142 project.

well since it reduces the amount of requests that should be handled by them. These advantages led to the concept of Content Distribution Network (CDN). CDN's distribute content on behalf of origins web servers in order to offload work from these servers. Although CDN's differ in their implementation model (e.g. distribution and redirection techniques) [11], they all take advantage of web caching in order to bring the content close to the clients.

A common and widespread technique that is used by CDN's to distribute their content is to push objects from web servers into web caches using a multicast-based channel. Using a broadcast channel such as satellite link, a CDN can push an object into many caches in one transmission instead of many unicast (terrestrial) transmissions. In this kind of a model web caches receive requests from clients (or other caches). If a request cannot be satisfied locally (i.e. there is no valid copy of the required object) the cache sends the request to the origin server. Then, the server sends the object to the cache using unicast transmission or it can distribute it to all caches using its multicast channel

The above mentioned CDN model is an example of a network application that uses both a multicast channel and unicast dissemination. This hybrid scheme is used by many other network applications that distribute contents and data to a large number of clients. The multicast channel is a limited resource and the utilization of the channel is an important issue. Determining which data should be sent by multicast and which data should be sent by unicast is a big challenge.

Other examples of network applications that may use this kind of hybrid scheme are reliable multicast protocols, real-time streaming protocols, and a context distribution systems. In a reliable multicast protocol (see, e.g., [12,8]) the data must be received by all clients. When a client does not receive a packet, it sends a feedback to the source that must retransmit this packet. The retransmissions of lost packets are critical since they may prevent clients from receiving other packets (according to the client receive window) and may cause timeout events. A real-time streaming protocol (see, e.g., [13]) is used to send multimedia streams to a set of clients. In this application, retransmission of lost packets is not mandatory and clients can use fractional data. However, retransmissions can be done to improve quality but these retransmissions are subject to time constraints since delayed data is useless. In both applications the multicast channel is used by the source to distribute the original transmissions while lost packets may be retransmitted using the multicast channel or the unicast dissemination. In a Context Distribution System pre-defined items that have time characteristic and time restriction, are distributed from producers to consumers. In [6] the authors study the efficiency of such a distribution using both a multicast channel and unicast dissemination.

In this paper we study the problem of utilizing a multicast channel in this hybrid scheme. In particular we concentrate on the scheduling algorithm that determines which data is sent via multicast channel at any given time. The problem of multicast scheduling has been discussed before in many papers. In [2,9] the authors present the problem of scheduling objects over a broadcast based channel. In both [9] and [2] the broadcast channel is the only available channel

and the objective function is to minimize the average response time (i.e. the period between the arrival of a request to the time in which this request is satisfied). In this kind of a model the scope of the cost function is very limited since it is proportional to the number of the requesters and the waiting time. Both papers focus on the on-line problem and present on-line scheduling algorithms for their problem. Acharya and Muthukrishnan [1] extend these results by a preemption mechanism that allows to interrupt a transmission and serve other requests before resuming an interrupted transmission. Moreover, additional objective functions are analyzed. Nevertheless, these cost functions still depend on the service time and they are proportional to the number of requesters and the waiting time. Su and Tassiulas [14] analyze the off-line problem of broadcast scheduling and formulate it as an optimization problem using a similar objective function (i.e. minimize the average response time).

The off-line problem of multicast scheduling in a hybrid scheme is defined in [5]. In this paper the authors describe this scheduling problem as an optimization problem in which the goal is to maximize the utilization of the multicast channel. In order to formulate the problem they have defined a benefit function that determines the amount of benefit one can get when a specific object is sent over the multicast channel at a specific time. This benefit function differs from one application to another and it depends on the objective function (e.g. minimizing the delay, minimizing the total transmissions, reducing the utilization of the unicast dissemination) and on other parameters such as time constraints attached to the specific model. They have shown an algorithm that solves the problem in polynomial time for the case where the objects are of fixed size. In addition, a 2-approximation algorithm was presented for the general case.

Nevertheless, in the model used in [5], an object can be sent by multicast only once. Such a restricted model cannot be used to accurately describe the scheme we presented. In web page distribution pages may have an expiration date, and a page may be updated after this time. In reliable multicast and streaming a packet can be lost again, and thus there may be a positive benefit from broadcasting it again. In this paper we present an extended model for the same hybrid scheme, in which the benefit function is more general. In particular, the benefit function depends on previous transmissions of the same object, as it can be sent multiple times. This scheduling problem is much more complex since the benefit at a specific time depends on the previous transmission of an object.

The problem is defined in Section 2. In Section 3 we present a non-trivial transformation that maps this problem to a special case of *maximum independent set*. Using this mapping we devise a 4-approximation algorithm for the case in which the objects have fixed size. In Section 4 we examine the general case in which the objects are of a variable size, and present a 10-approximation algorithm for this case. Both algorithms are based on the *local ratio technique* [4]. In Section 5 we study a simple greedy algorithm. We show that even though in general the approximation ratio of this algorithm is unbounded, if the benefit function follows certain restrictions, which are of practical interest, it is bounded by 3. A discussion related to the hardness of the problem (both in the off-line

and on-line cases) and the proof of the 10-approximation algorithm presented in Section 4, is left for the full version of this paper. In addition we also study the case where  $k > 1$  multicast channels are available, and we obtain approximation algorithms with the same performance guarantees in this case.

## 2 Model and Problem Definition

In the off-line problem the sequence of requests is known in advance, namely the source determines its scheduling according to full information. This kind of a problem takes place in applications such as Information Notification Services where the clients specify their requirement in advance. In other applications such as CDN or reliable multicast protocols the sequence of requests is obtained during the execution of the scheduling. In order to employ an off-line scheduling algorithm in such models one should predict the sequence of requests. For instance the sequence can be predicted using statistical information gathered during previous periods.

First we restrict the discussion to the case where all objects have the same size. We divide the time axis into discrete time slots, such that it takes one time slot to broadcast one object. In this case a scheduling algorithm determines which object should be sent via multicast in each time slot. Later on in Section 4 we discuss the general case, where different objects may have different sizes.

### 2.1 The Benefit Function

The benefit function describes the benefit of sending a specific object via the multicast channel at a specific time. This benefit is the difference between the benefit one can get with and without the multicast channel and it depends on the objective function of each application. The benefit of sending an object at a specific time depends on the requisiteness of the object before and after that time. Therefore, the exact computation of the benefit function requires knowledge regarding the full sequence of requests, and it can be computed and used only in the off-line problem. In addition, the benefit function depends on the last time at which the object has been sent, since this previous transmission updates all clients and affects their need for a new copy. According to these observations we define the following benefit function.

**Definition 1 (Benefit Function).** *The benefit function  $B_{t,t',j}$  is the benefit of broadcasting object  $p_j$  at time slot  $t$  where  $t'$  was the last time (before  $t$ ) where  $p_j$  was broadcasted. In addition,  $B_{t,0,j}$  is the benefit of broadcasting object  $p_j$  at time slot  $t$ , where  $t$  is the first time slot where  $p_j$  was broadcasted. Otherwise (i.e. where  $t' \geq t$ )  $B_{t,t',j} = 0$ .*

It is very convenient to represent the benefit function by  $m$  matrices of size  $(T+1) \times (T+1)$  where  $m$  is the number of objects and  $T$  is the number of time slots in the period. In this kind of representation, the benefit function of each object is represented by one matrix, and the elements in the matrix describe the

benefit function of the object in the following way:  $B_{t,t',j}$  is the element in column  $t$  and row  $t'$  of matrix  $j$ . Note that according to Definition 1,  $t' < t$ , and therefore the benefit matrices are upper diagonal. For instance, Figure 1 depicts a benefit function of two objects for a period of 3 time slots. Assuming that the multicast channel is always available, one cannot lose by sending an object via multicast, hence the benefit function is non negative (i.e.  $\forall 0 \leq t' < t \leq T, j, B_{t,t',j} \geq 0$ ).

	<u>0</u> <u>1</u> <u>2</u> <u>3</u>			<u>0</u> <u>1</u> <u>2</u> <u>3</u>	
	0 <u>0</u> <u>5</u> <u>4</u> <u>3</u>			0 <u>0</u> <u>4</u> <u>5</u> <u>2</u>	
Object 1:	1 <u>0</u> <u>0</u> <u>8</u> <u>10</u>		Object 2:	1 <u>0</u> <u>0</u> <u>4</u> <u>3</u>	
	2 <u>0</u> <u>0</u> <u>0</u> <u>6</u>			2 <u>0</u> <u>0</u> <u>0</u> <u>5</u>	
	3 <u>0</u> <u>0</u> <u>0</u> <u>0</u>			3 <u>0</u> <u>0</u> <u>0</u> <u>0</u>	

**Fig. 1.** A Scheduling Example

For example, consider a CDN application that uses a satellite based channel as a multicast channel. The CDN server distributes objects that have an expiration date or a max-age character and its objective function is to reduce usage of the unicast dissemination, namely to reduce the amount of data that is sent via unicast. An expiration date of an object refers to a time at which the object should no longer be returned by a cache unless the cache gets a new (validated) version of this object from the server with a new expiration date. A max-age time of an object refers to the maximum time, since the last time this object has been acquired by a cache, at which the object should no longer be returned by the cache unless it gets a new (validated) version of this object from the server with a new max-age time (see [7, Section 13.2]). Given a set of objects and their sequence of requests for a period of  $T$  time slots, the benefit function can be easily built as follows: for each object  $p_j$  and for each time slot  $t$ , we calculate the benefit of broadcasting  $p_j$  at time slot  $t$ , assuming that  $t'$  was the last time slot before  $t$ , at which the object was broadcasted. This can be done (assuming that the sequence of request is known) by simulating the case where the object is not sent via multicast vs. the case in which the object is sent via multicast. For instance, consider an object  $p_j$  with max-age of five time slots and assume that this object is required by cache number one at time slots 2, 6, and 8. We calculate  $B_{5,1,j}$  as follows. If the object is broadcasted at time slots 1 and 5, it means that the object is not sent via unicast at all. On the other hand, if the object is not broadcasted at time slot 5 (i.e. only at time slot 1), it must be sent via unicast at time slot 8, namely  $B_{5,1,j}$  is equal to one transmission of  $p_j$ .

## 2.2 Problem Definition

The Time Dependent Multi Scheduling of Multicast problem (TDMSM) is defined as follow: Given a benefit function, find a feasible schedule with maximum benefit. A natural way to represent a schedule is by a function  $S$  where  $S(t)$  stands for the index of the object  $j$  that was sent at time slot  $t$ . ( $S$  may be undefined on part of the time slots.) The benefit of a schedule is the sum of the

benefit values that are derived from it, namely if  $t$  and  $t'$  are two consecutive transmission time slots of object  $p_j$ , then  $B_{t,t',j}$  is added to the total benefit. For instance, assume that object  $p_1$  is sent at time slot 2 and object  $p_2$  is sent at time slots 1 and 3. This schedule can be represented by:  $S(1) = 2, S(2) = 1$ , and  $S(3) = 2$ , and the total benefit of this schedule is  $B_{2,0,1} + B_{1,0,2} + B_{3,1,2}$ .

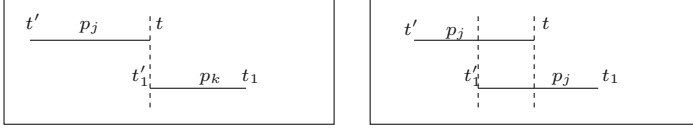
Another way to represent a schedule is by a set of triples, where each triple  $(t, t', j)$  contains the index of the object  $j$ , the time it was sent  $t$ , and the last time before  $t$  at which the object was sent  $t'$  (or 0 if the object was not sent before). In this kind of representation, the triple  $(t, t', j)$  is in the set if and only if  $t$  and  $t'$  are two consecutive transmissions of object  $p_j$  ( $t > t'$ ). For instance, the schedule described above can be represented by the following set of triples:  $\{(2, 0, 1), (1, 0, 2), (3, 1, 2)\}$ . Although the notation of triples seems to be unnatural and more complex, it has the advantage that the benefit values of are derived directly from the triples. In other words, each triple contains the time slots of two consecutive transmissions, therefore the triple  $(t, t', j)$  is in the set if and only if the benefit value  $B_{t,t',j}$  should be added to the total benefit (in the former representation, using a set of pairs, one should find a consecutive transmissions by parsing the whole set). Nevertheless, not every set of triples represent a feasible schedule. For instance if the triple  $(6, 4, 3)$  is in the set it indicates that object  $p_3$  was sent at time slots 4 and 6. Since other objects cannot be sent at these time slots, the triple  $(7, 6, 9)$  cannot be in the set (i.e. object  $p_9$  cannot be sent at time slot 6). Moreover the triple  $(6, 4, 3)$  indicates that time slots 4 and 6 are two consecutive transmissions of object  $p_3$ , namely it was not sent at time slot 5, therefore the triple  $(8, 5, 3)$  cannot be in the set. On the other hand it is clear that every schedule can be represented by a set of triples (every two consecutive transmissions of an object are a triple in the set). In Section 3 we use the notation of triples and the correlation between triples and a benefit values to represent the scheduling problem as an independent set problem and to present efficient scheduling algorithm that are based on this representation.

### 3 A 4-Approximation Algorithm

In Section 2.2 we showed that a feasible schedule can be represented by a set of triples, where each triple  $(t, t', j)$  contains the index of the object  $j$ , and the time slots of two consecutive transmissions of this object:  $t$  and  $t'$  ( $t > t'$ ). In this section we use this notation and the correlation between triples and benefit values (i.e. the triple  $(t, t', j)$  is in the set if and only if the benefit value  $B_{t,t',j}$  is added to the total benefit) to devise a 4-approximation algorithm to TDMSM. The algorithm uses the local-ratio technique [4] and is based on the representation of TDMSM as special case of the *maximum weight independent set* problem.

It is very convenient to represent a triple  $(t, t', j)$  as an interval, where the end point of the interval are  $t$  and  $t'$ . Using this kind of representation we can describe the relations between triples according to their location of the corresponding intervals. We say that a triple  $(t, t', j)$  *intersects* a triple  $(t_1, t'_1, j)$  (where  $t > t'$  and  $t_1 > t'_1$ ) if the intervals  $(t', t)$  and  $(t'_1, t_1)$  intersect. In other words, the triples

intersect if  $t'_1 < t$  and  $t_1 > t'$  (The right part of Figure 2 depicts the intervals of two triples  $(t, t', j)$  and  $(t_1, t'_1, j)$  that intersect). A triple  $(t, t', j)$  *overlaps* a triple  $(t_1, t'_1, k)$  (where  $t > t'$ ,  $t_1 > t'_1$  and  $j \neq k$ ) if the intervals  $(t', t)$  and  $(t'_1, t_1)$  have at least one common end point that is not zero (i.e.  $t = t_1$  or  $t' = t'_1$  or  $t = t'_1$  or  $t' = t'_1 \neq 0$ ) (The left part of Figure 2 depicts the intervals of two triples  $(t, t', j)$  and  $(t_1, t'_1, k)$  that overlap). A set of triples is called *feasible* if it does not contain overlapping triples or intersecting triples. A feasible set of triples  $S$  is called *maximal* if  $\forall (t, t', j) \notin S$ , the set  $S' = \{(t, t', j)\} \cup S$  is not feasible (i.e. one cannot add a triple to the set without turning it into an infeasible set).



**Fig. 2.** Overlapping and intersecting triples.

**Lemma 1.** *A set of triples that describes a feasible schedule is feasible.*

*Proof.* Consider a feasible schedule in which an object  $p_j$  is sent at time slots  $t_{j_1}, \dots, t_{j_n}$ . The scheduling of object  $p_j$  is represented by the following set of triples  $\{(t_{j_1}, 0, j), (t_{j_2}, t_{j_1}, j), \dots, (t_{j_n}, t_{j_{n-1}}, j)\}$ , therefore it does not contain intersecting triples. Also, in a feasible schedule only one object can be sent every time slot, namely object  $p_i$  ( $i \neq j$ ) cannot be sent at time slots  $t_{j_1}, \dots, t_{j_n}$ , therefore it does not contain overlapping triples.  $\square$

**Lemma 2.** *A maximal set of triples  $S$  induces a feasible schedule.*

*Proof.* Since  $S$  does not contain overlapping triples, at most one object is sent at every time slot. Since  $S$  does not contain intersecting triples, if  $(t, t', j) \in S$ , then  $t$  and  $t'$  are two consecutive transmissions of  $p_j$ . The maximality requirement ensures that if  $t$  and  $t'$  are two consecutive broadcastings then  $(t, t', j) \in S$ .  $\square$

Given a benefit function, we can construct a weighted graph  $G = (V, E)$ , in which every triple  $(t, t', j)$  is represented by a vertex  $v_{t,t',j} \in V$  whose weight is  $w(v_{t,t',j}) = B_{t,t',j}$ . An edge  $(v_{t,t',j}, v_{t_1,t'_1,k}) \in E$  if  $(t, t', j)$  intersects or overlaps  $(t_1, t'_1, k)$ . Henceforth we say that  $v_{t,t',j}$  intersects (overlaps)  $v_{t_1,t'_1,k}$  if the triple  $(t, t', j)$  intersects (overlaps) the triple  $(t_1, t'_1, k)$ .

By the construction of  $G$ , finding a maximum and maximal independent set is equivalent to finding a solution to TDMSM. Finding a maximal independent set implies a feasible schedule, since an independent set does not contain adjacent vertices, namely the corresponding set of triples does not contain overlapping or intersecting triple. Moreover, the weight of the independent set is derived from the weight of the vertices in the set, which are equivalent to the benefit values in the schedule. Similarly, a feasible schedule does not contain intersecting and overlapping triples, therefore the corresponding vertices in  $G$  are independent set, and the benefit of the schedule is the same as the weight of the set.

In general, maximum independent set is hard to approximate [10]. Nevertheless, the graph that is derived from a TDMSM instance is a special case, in which the edges of the graph are characterized by the intersecting and overlapping notation. We describe a recursive algorithm that finds an independent set in  $G$ , and we prove that the algorithm is 4-approximation. We note that unlike other scheduling problems that were approximated using this kind of technique (see, e.g., [3]), the output of our algorithm does not induce a feasible schedule. Namely, our algorithm finds an independent set and not necessarily a maximal independent set. To obtain a feasible schedule one should extend the output of the algorithm to a maximal independent set. Since the benefit function is non-negative, the benefit of the extended solution is not less than the benefit of the original independent set. Thus, the corresponding schedule is 4-approximate.

**Algorithm  $\mathcal{IS}(G = (V, E), w)$**

1. If  $V = \emptyset$  return  $\emptyset$ .
2. Choose  $v_{t,t',j} \in V$ , such that  $\forall v_{t_1,t'_1,k}, t \leq t_1$ .
3.  $\forall v_{t_1,t'_1,k} \in V, w_1(v_{t_1,t'_1,k}) = \begin{cases} w(v_{t,t',j}) & (v_{t,t',j}, v_{t_1,t'_1,k}) \in E \\ w(v_{t,t',j}) & v_{t_1,t'_1,k} \equiv v_{t,t',j} \\ 0 & \text{otherwise} \end{cases}$
4. For all  $v_{t_1,t'_1,k} \in V, w_2(v_{t_1,t'_1,k}) = w(v_{t_1,t'_1,k}) - w_1(v_{t_1,t'_1,k})$ .
5. Denote  $V' = \{v_{t_1,t'_1,k} \mid w_2(v_{t_1,t'_1,k}) \leq 0\}$ .
6.  $\tilde{V} = V \setminus V'$ .
7.  $\tilde{S} = \mathcal{IS}(\tilde{G} = (\tilde{V}, E), w_2)$ .
8. If  $\tilde{S} \cup \{v_{t,t',j}\}$  is an independent set  $S = \tilde{S} \cup \{v_{t,t',j}\}$
9. Else  $S = \tilde{S}$ .
10. return  $S$

The number of vertices in the graph depends on the number of objects and the number of time slots, namely  $|V| = \frac{1}{2}mT(T-1)$ . In Line 2 the algorithm picks a vertex  $v_{t,t',j}$  such that  $t$  is minimal. If more than one vertex can be selected, it is efficient to pick the vertex with maximum weight. There are  $O(T)$  iterations since the minimal  $t$  increases in every iteration. In a naive implementation every iteration requires  $O(mT^2)$  operations (finding the maximum weight, subtracting weights, and removing vertices) thus the complexity of the algorithm is  $O(mT^3)$ .

Next we prove that Algorithm  $\mathcal{IS}$  is a 4-approximation algorithm using the Local Ratio Theorem for maximization problem (see [3] for more details).

**Theorem 1 (Local Ratio [3]).** *Let  $F$  be a set of constraints and let  $p, p_1$  and  $p_2$  be a benefit functions such that  $p = p_1 + p_2$ . If  $x$  is  $r$ -approximation solution with respect to  $p_1$  and  $p_2$  then  $x$  is  $r$ -approximation with respect to  $p$ .*

In our context the benefit functions are  $w, w_1$  and  $w_2$  (according to Line 4,  $w = w_2 + w_1$ ). The constraints are that the set (of vertices) is independent.

**Lemma 3.** *The weight of a maximum independent set with respect to  $(G, w_1)$  is at most  $4 \cdot w_1(v_{t,t',j})$ .*



*Proof.* By construction of  $w_1$  vertices that overlap or intersect  $v_{t,t',j}$  (including  $v_{t,t',j}$ ) are given a weight of  $w_1(v_{t,t',j})$  (Line 3). All other vertices have a weight of zero and their contribution to the weight of the independent set is null. Therefore, we consider only vertices that intersect or overlap  $v_{t,t',j}$ . We divide these vertices into two sets. Denote by  $I(v_{t,t',j})$  the set of vertices that intersect  $v_{t,t',j}$  including  $v_{t,t',j}$  and by  $O(v_{t,t',j})$  the set of vertices that overlap  $v_{t,t',j}$  including  $v_{t,t',j}$ .

Consider two vertices  $v_{t_1,t'_1,j}, v_{t_2,t'_2,j} \in I(v_{t,t',j})$ . Both vertices intersect  $v_{t,t',j}$ , namely  $t'_1 < t$  and  $t'_2 < t$ . Also,  $t \leq t_1$  and  $t \leq t_2$  since  $t$  is minimum (Line 2). Thus,  $v_{t_1,t'_1,j}$  and  $v_{t_2,t'_2,j}$  intersect. This means that  $I(v_{t,t',j})$  induces a clique, and hence any independent set cannot contain more than one vertex from  $I(v_{t,t',j})$ .

Observe that any  $v_{t_1,t'_1,k} \in O(v_{t,t',j})$  satisfies  $t_1 = t, t'_1 = t$ , or  $t'_1 = t'$  ( $t_1 = t'$  is not possible since  $t$  is minimum). Thus we divide  $O(v_{t,t',j})$  into three subsets:

$$\begin{aligned} O_1(v_{t,t',j}) &= \{v_{t_1,t'_1,k} \mid v_{t_1,t'_1,k} \in O(v_{t,t',j}) \wedge (t_1 = t)\} \\ O_2(v_{t,t',j}) &= \{v_{t_1,t'_1,k} \mid v_{t_1,t'_1,k} \in O(v_{t,t',j}) \wedge (t'_1 = t)\} \\ O_3(v_{t,t',j}) &= \{v_{t_1,t'_1,k} \mid v_{t_1,t'_1,k} \in O(v_{t,t',j}) \wedge (t_1 = t')\} \end{aligned}$$

Consider two vertices  $v_{t_1,t'_1,k}, v_{t_2,t'_2,l} \in O_1(v_{t,t',j})$ . If  $k = l$  the vertices intersect (since  $t_1 < t$  and  $t'_1 < t$ ), otherwise the vertices overlap. Thus,  $O_1(v_{t,t',j})$  induces a clique, and an independent set cannot contain more than one vertex from  $O_1(v_{t,t',j})$ . Similarly, it can be shown that  $O_2(v_{t,t',j})$  and  $O_3(v_{t,t',j})$  induce cliques. Hence, an independent set cannot contain more than one vertex from  $O_2(v_{t,t',j})$  and one vertex from  $O_3(v_{t,t',j})$ .

Putting it all together, an independent set contains at most four vertices. The weight of each vertex is  $w_1(v_{t,t',j})$ , thus its maximum weight is  $4 \cdot w_1(v_{t,t',j})$ .  $\square$

**Theorem 2.** *Algorithm  $\mathcal{IS}$  is a 4-approximation algorithm.*

*Proof.* The proof is by induction on the recursion. At the induction base (Line 1 of  $\mathcal{IS}$ )  $V = \emptyset$ , and the algorithm returns an optimal solution. For the induction step, we assume that  $\tilde{S}$  is 4-approximate with respect to  $(\tilde{G}, w_2)$  (Line 7). We show that  $S$  is 4-approximate with respect to  $(G, w_2)$  and  $(G, w_1)$ . This proves, by Local Ratio Theorem, that  $S$  is 4-approximate with respect to  $(G, w)$ .

Due to Line 5  $\bar{V}$  contains vertices with non-positive weight, thus the optimum with respect to  $(G, w_2)$  is not greater than the optimum with respect to  $(\tilde{G}, w_2)$ . Hence  $\tilde{S}$  is 4-approximate with respect to  $(G, w_2)$ . Due to Lines 3 and 4,  $w_2(v_{t,t',j}) = 0$ , therefore adding  $v_{t,t',j}$  to  $S$  (in Line 8) does not reduce the weight of the independent set. Hence  $S$  is 4-approximate with respect to  $(G, w_2)$ .

$S$  contains  $v_{t,t',j}$  or at least one of its neighbors (otherwise  $v_{t,t',j}$  have been added to  $S$  in Line 8). Hence, the weight of  $S$  is at least  $w_1(v_{t,t',j})$  with respect to  $(G, w_1)$  where the maximum independent set is  $4 \cdot w_1(v_{t,t',j})$  (see Lemma 3), namely  $S$  is 4-approximate with respect to  $(G, w_1)$ .  $\square$

## 4 Variable Size Objects

In the general case, where different objects may have different sizes we divide the time axis such that the transmission of one object requires one or more integral

time slots. One way to satisfy this requirement is to find the greater common divider of the objects' sizes and use this value as the length of the time slot. We denote by  $\tau_j$  the number of time slots it takes to broadcast  $p_j$ . For instance, if the transmission times of  $p_1$ ,  $p_2$  and  $p_3$  are 0.4, 1 and 1.4 millisecond, respectively, the length of the time slot can be 0.2 millisecond and  $\tau_1 = 2$ ,  $\tau_2 = 5$  and  $\tau_3 = 7$ .

The benefit function and the problem definition in this case are similar to those defined in Section 2. However, the different sizes of the objects induce some changes that are reflected in the feasibility of the schedule and in the exact definition of the benefit function. When  $p_j$  is sent at time slot  $t$  the subsequent  $\tau_j$  time slots are occupied by this transmission. Thus, in a feasible schedule these time slots cannot be used for other transmissions. Also, if  $t$  and  $t'$  are two consecutive transmissions of  $p_j$ ,  $t$  cannot be smaller than  $t' + \tau_j$ , namely the gap between two consecutive transmissions is at least  $\tau_j$ . Hence,  $\forall t' > 0, t \leq t' + \tau_j$ ,  $B_{t,t',j} = 0$ . Note that in Section 2.1, where  $\forall j$ ,  $\tau_j = 1$ , we implicitly considered this restriction by defining the benefit function to be 0 where  $t' \geq t$ .

In the full version of the paper we show a 10-approximation algorithm to TDMSM using the same technique that has been used in Section 3. This algorithm is a modified version of Algorithm  $\mathcal{IS}$  in which in every iteration we consider vertices that correspond to objects with a minimal transmission time, namely the following steps should replace Line 2 in  $\mathcal{IS}$ :

- 2a. Choose  $j$  such that  $\forall k$ ,  $\tau_j \leq \tau_k$
- 2b. Choose  $v_{t,t',j} \in V$ , such that  $\forall v_{t_1,t'_1,j}$ ,  $t \leq t_1$

## 5 An Efficient Algorithm for a Special Case

The approximation results described so far hold for any non-negative benefit function. However, when using this scheme in practical scenarios, the benefit function may obey several non-trivial limitations. In this section we consider certain limitations on the benefit function, and we show that in this case a greedy algorithm approximates the optimal solution by a factor of 3.

Consider a scenario in which a source sends object  $p_j$  via multicast at two consecutive time slots  $t_1$  and  $t_3$ , where  $t_3 > t_1$ . The benefit of such a schedule is  $B_{t_3,t_1,j}$ . If the source transmits  $p_j$  also at time slot  $t_2$ , where  $t_1 < t_2 < t_3$ , then the total benefit that of these transmissions is  $B_{t_3,t_2,j} + B_{t_2,t_1,j}$ . In general, this additional transmission may decrease the total benefit (i.e.,  $B_{t_3,t_2,j} + B_{t_2,t_1,j} < B_{t_3,t_1,j}$ ). However, this does not happen in practice. Recall that  $B_{t_3,t_1,j}$  represents the benefit of broadcasting  $p_j$  at time slot  $t_3$  given that the previous multicast transmission was performed at time slot  $t_1$ . Therefore, a multicast transmission at  $t_2$  should not add unicast transmissions of  $p_j$ . Therefore, we may assume that for any object  $p_j$ , and time slots  $t_1 < t_2 < t_3$  the following *triangle inequality* holds:  $B_{t_3,t_2,j} + B_{t_2,t_1,j} \geq B_{t_3,t_1,j}$ . Notice that due to the triangle inequality we may assume that there exists an optimal solution that uses the multicast channel in every time slot.

Next we consider a restriction that indicates that benefit cannot appear suddenly. Assume that a source sends  $p_j$  via multicast at time slots  $t_2$  and  $t_3$  where

$t_3 > t_2$ . The benefit of such schedule is  $B_{t_3, t_2}^j$ . In many practical scenarios this benefit comes from the fact that the object was sent at  $t_3$  (regardless of the previous time it was sent) and from the fact that it was already sent at  $t_2$ . Thus, for any time slot  $t_1$  such that  $t_1 < t_2$  the following *skewed triangle inequality* should hold  $B_{t_2, t_1, j} + B_{t_3, t_1, j} \geq B_{t_3, t_2, j}$ . In the full version we show that TDMSM remains NP-hard even under both restrictions.

Recall that a schedule  $S$  is a function, where  $S(t)$  is the index  $j$  of the object that was sent at time slot  $t$ . Let  $P_S(j, t)$  be the time slot before time  $t$  in which object  $j$  was last broadcasted in solution  $S$ . If object  $j$  was not broadcasted before time  $t$  we define  $P_S(j, t) = 0$ . Also, let  $N_S(j, t)$  be the first time slot after time  $t$  in which object  $j$  was broadcasted in solution  $S$ . If object  $j$  was not broadcasted after time  $t$  we define  $N_S(j, t) = T + 1$ . We denote by  $B(S)$  the benefit gained by the solution  $S$ .

Next we present our greedy algorithm.

**Algorithm Greedy**

1. for  $t = 1$  to  $T$  do
2.      $\ell \leftarrow \operatorname{argmax}_j \{B_{t, P(j, t), j}\}$
3.     Broadcast  $p_\ell$

For general benefit functions **Greedy** does not guarantee any approximation ratio. For instance, consider the following benefit function:

	$\begin{array}{c ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & \epsilon & 0 \\ 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{array}$		$\begin{array}{c ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 2\epsilon & 0 \\ 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{array}$
Object 1:		Object 2:	

In this case the benefit of the optimal schedule is  $B_{1,0,1} + B_{2,1,1} = 1 + \epsilon$ , while the benefit of the schedule returned by the algorithm is  $B_{2,0,2} + B_{2,1,2} = 2\epsilon$ . Nevertheless, we show that if the benefit function follows the above mentioned restrictions, Algorithm **Greedy** returns 3-approximate solutions.

We denote by  $G$  the greedy solution and by  $\text{Opt}$  an optimal solution. We also define a series of hybrid solutions denoted by  $H_0, \dots, H_T$ , where

$$H_t(i) = \begin{cases} G(i) & i \leq t, \\ \text{Opt}(i) & i > t. \end{cases}$$

Note that  $H_0 = \text{Opt}$  and  $H_T = G$ . We now examine the difference in benefit gained by  $H_{t-1}$  and by  $H_t$ . Denote by  $\Delta_t$  the difference in benefit gained by  $H_{t-1}$  and  $H_t$ , that is  $\Delta_t = B(H_{t-1}) - B(H_t)$ . We show  $\Delta_t \leq 2 \cdot B_{t, P_G(G(t), t), G(t)}$ . Let  $j = \text{Opt}(t)$  and  $\ell = G(t)$ . Then,

$$\begin{aligned} \Delta_t &= B_{t, P_{H_t}(j, t), j} + B_{N_{H_t}(j, t), t, j} - B_{N_{H_t}(j, t), P_{H_t}(j, t), j} \\ &\quad - B_{t, P_{H_t}(\ell, t), \ell} - B_{N_{H_t}(\ell, t), t, \ell} + B_{N_{H_t}(\ell, t), P_{H_t}(\ell, t), \ell}. \end{aligned}$$

By the triangle inequality  $B_{N_{H_t}(\ell,t),P_{H_t}(\ell,t),\ell} \leq B_{N_{H_t}(\ell,t),t,\ell} + B_{t,P_{H_t}(\ell,t),\ell}$ . Thus,

$$\Delta_t \leq B_{t,P_{H_t}(j,t),j} + B_{N_{H_t}(j,t),t,j} - B_{N_{H_t}(j,t),P_{H_t}(j,t),j}.$$

By the skewed triangle inequality,  $B_{N_{H_t}(j,t),t,j} - B_{N_{H_t}(j,t),P_{H_t}(j,t),j} \leq B_{t,P_{H_t}(j,t),j}$  which means that  $\Delta_t \leq 2B_{t,P_{H_t}(j,t),j}$ .  $H_t(i) = G(i)$  for every  $i < t$  by the definition of  $H_t$ , thus  $\Delta_t \leq 2B_{t,P_G(j,t),j}$ .  $B_{t,P_G(j,t),j} \leq B_{t,P_G(\ell,t),\ell}$  since **Greedy** picks the object that maximizes the benefit, and therefore  $\Delta_t \leq 2B_{t,P_G(\ell,t),\ell}$ .

**Greedy** is a 3-approximation algorithm since  $B(G) \geq \frac{1}{3} \cdot B(\text{Opt})$ :

$$B(\text{Opt}) - B(G) = B(H_0) - B(H_T) = \sum_{t=1}^T \Delta_t \leq 2 \cdot \sum_{t=1}^T B_{t,P_{H_t}(G(t),t),G(t)} = 2B(G).$$

## References

1. S. Acharya and S. Muthukrishnan. Scheduling on-demand broadcasts: New metrics and algorithms. In *Mobile Computing and Networking*, pages 43–54, 1998.
2. D. Aksoy and M. Franklin.  $R \times W$ : a scheduling approach for large-scale on-demand data broadcast. *IEEE/ACM Transactions on Networking*, 7(6):846–860, 1999.
3. A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Shieber. A unified approach to approximating resource allocation and scheduling. *Journal of the ACM*, 48(5):1069–1090, 2001.
4. R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, 25:27–46, 1985.
5. R. Cohen, L. Katsir, and D. Raz. Scheduling algorithms for a cache pre-filling content distribution network. In *21st Annual Joint Conference of the IEEE Computer and Communications Society*, volume 2, pages 940–949, 2002.
6. R. Cohen and D. Raz. An open and modular approach for a context distribution system. In *9th IEEE/IFIP Network Operations and Management Symposium (NOMS 2004)*, pages 365–379, 2004.
7. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2186: Hypertext Transfer Protocol – HTTP/1.1, June 1999.
8. S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.
9. S. Hameed and N. H. Vaidya. Log-time algorithms for scheduling single and multiple channel data broadcast. In *Mobile Computing and Networking*, pages 90–99, 1997.
10. J. Håstad. Clique is hard to approximate within  $n^{1-\epsilon}$ . In *37th IEEE Symposium on Foundations of Computer Science*, pages 627–636, 1996.
11. B. Krishnamurthy, C. Wills, and Y. Zhang. On the use and performance of content distribution networks. In *Proceedings of the 1st ACM SIGCOMM Internet Measurement Workshop (IMW-01)*, pages 169–182, 2001.
12. S. Paul, K. K. Sabnani, J. C. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (rmtp). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, 1997.
13. H. Schulzrinne, A. Rao, and R. Lanphier. RFC 2326: Real time streaming protocol (RTSP), Apr. 1998.
14. C.-J. Su and L. Tassiulas. Broadcast scheduling for information distribution. In *INFOCOM (1)*, pages 109–117, 1997.
15. D. Wessels and K. Claffy. RFC 2186: Internet Cache Protocol (ICP), version 2, Sept. 1997.

# Convergence Properties of the Gravitational Algorithm in Asynchronous Robot Systems

Reuven Cohen and David Peleg

Department of Computer Science and Applied Mathematics,  
The Weizmann Institute of Science, Rehovot 76100, Israel  
{r.cohen,david.peleg}@weizmann.ac.il

**Abstract.** This paper considers the convergence problem in autonomous mobile robot systems. A natural algorithm for the problem requires the robots to move towards their center of gravity. Previously it was known that the gravitational algorithm converges in the synchronous or semi-synchronous model, and that two robots converge in the asynchronous model. The current paper completes the picture by proving the correctness of the gravitational algorithm in the fully asynchronous model for any number of robots. It also analyses its convergence rate, and establishes its convergence in the presence of crash faults.

## 1 Introduction

### 1.1 Background and Motivation

Swarms of low cost robots provide an attractive alternative when facing various large-scale tasks in hazardous or hostile environments. Such systems can be made cheaper, more flexible and potentially resilient to malfunction. Indeed, interest in autonomous mobile robot systems arose in a variety of contexts (see [4,5,14,15,16,17,18,19,20,21,28] and the survey in [6,7]).

Along with developments related to the physical engineering aspects of such robot systems, there have been recent research attempts geared at developing suitable algorithmics, particularly for handling the distributed coordination of multiple robots [3,8,9,22,24,26,27]. A number of computational models were proposed in the literature for multiple robot systems. In this paper we consider the fully asynchronous model of [8,9,12,23]. In this model, the robots are assumed to be identical and indistinguishable, lack means of communication, and operate in Look-Compute-Move cycles. Each robot wakes up at unspecified times, observes its environment using its sensors (capable of identifying the locations of the other robots), performs a local computation determining its next move and moves accordingly.

Much of the literature on distributed control algorithms for autonomous mobile robots has concentrated on two basic tasks, called *gathering* and *convergence*. Gathering requires the robots to occupy a single point within finite time, regardless of their initial configuration. Convergence is the closely related task in which the robots are required to converge to a single point, rather than reach it. More

precisely, for every  $\epsilon > 0$  there must be a time  $t_\epsilon$  by which all robots are within a distance of at most  $\epsilon$  of each other.

A common and straightforward approach to these tasks relies on the robots in the swarm calculating some median position and moving towards it. Arguably the most natural variant of this approach is the one based on using the *center of gravity* (sometimes called also the *center of mass*, the *barycenter* or the average) of the robot group. This approach is easy to analyze in the synchronous model. In the asynchronous model, analyzing the process becomes more involved, since the robots operate at different rates and may take measurements at different times, including while other robots are in movement. The inherent asynchrony in operation might therefore cause various oscillatory effects on the centers of gravity calculated by the robots, preventing them from moving towards each other and possibly even cause them to diverge and stray away from each other in certain scenarios.

Several alternative, more involved, algorithms have been proposed in the literature for the gathering and convergence problems. The gathering problem was first discussed in [26,27] in a semi-synchronous model, where the robots operate in cycles but not all robots are active in every cycle. It was proven therein that it is impossible to gather *two* oblivious autonomous mobile robots without a common orientation under the semi-synchronous model (although 2-robot convergence is easy to achieve in this setting). On the other hand, there is an algorithm for gathering  $N \geq 3$  robots in the semi-synchronous model [27]. In the asynchronous model, an algorithm for gathering  $N = 3, 4$  robots is presented in [9,23], and an algorithm for gathering  $N \geq 5$  robots has recently been described in [8]. The gathering problem was also studied in a system where the robots have limited visibility [2,13]. Fault tolerant algorithms for gathering were studied in [1]. In a failure-prone system, a gathering algorithm is required to successfully gather the nonfaulty robots, independently of the behavior of the faulty ones. The paper presents an algorithm tolerant against a single crash failure in the asynchronous model. For Byzantine faults, it is shown therein that in the asynchronous model it is impossible to gather a 3-robot system, even in the presence of a single Byzantine fault. In the fully synchronous model, an algorithm is provided for gathering  $N$  robots with up to  $f$  faults, where  $N \geq 3f + 1$ .

Despite the existence of these elaborate gathering algorithms, the gravitational approach is still very attractive, for a number of reasons. To begin with, it requires only very simple and efficient calculations, which can be performed on simple hardware with minimal computational efforts. It can be applied equally easily to any number of dimensions and to any swarm size. Moreover, the errors it incurs due to rounding are bounded and simple to calculate. In addition, it is oblivious (i.e., it does not require the robots to store any information on their previous operations or on past system configurations). This makes the method both memory-efficient and self-stabilizing (meaning that following a finite number of transient errors that change the states of some of the robots into other (possibly illegal) states, the system returns to a legal state and achieves eventual convergence). Finally, the method avoids deadlocks, in the sense that every

robot can move at any given position (unless it has already reached the center of gravity). These advantages may well make the gravitational algorithm the method of choice in many practical situations.

Subsequently, it is interesting to study the correctness and complexity properties of the gravitational approach to convergence. This study is the focus of the current paper. Preliminary progress was made by us in [10], where we proved the correctness of the gravitational algorithm in the semi-synchronous model of [26]. In the asynchronous model, we provided a convergence proof for the special case of a 2-robot system. In the current paper, we complete the picture by proving a general theorem about the convergence of the center of gravity algorithm in the fully asynchronous model. We also analyze the convergence rate of the algorithm. Finally, we establish convergence in the crash fault model. Specifically, we show that in the presence of  $f$  crash faults,  $1 \leq f \leq N - 2$ , the  $N - f$  nonfaulty robots will converge to the center of gravity of the crashed robots.

## 1.2 The Model

The basic model studied in [3,8,9,22,24,26,27] can be summarized as follows. The  $N$  robots execute a given algorithm in order to achieve a prespecified task. Each robot  $i$  in the system operates individually, repeatedly going through simple cycles consisting of three steps:

- **Look:** Identify the locations of all robots in  $i$ 's private coordinate system and obtain a multiset of points  $P = \{p_1, \dots, p_N\}$  defining the current *configuration*. The robots are indistinguishable, so  $i$  knows its own location  $p_i$  but does not know the identity of the robots at each of the other points. This model allows robots to detect multiplicities, i.e., when two or more robots reside at the same point, all robots will detect this fact. Note that this model is stronger than, e.g., the one of [8].
- **Compute:** Execute the given algorithm, resulting in a goal point  $p_G$ .
- **Move:** Move on a straight line towards the point  $p_G$ . The robot might stop before reaching its goal point  $p_G$ , but is guaranteed to traverse a distance of at least  $S$  (unless it has reached the goal). The value of  $S$  is not assumed to be known to the robots, and they cannot use it in their calculations.

The Look and Move operations are identical in every cycle, and the differences between various algorithms are in the Compute step. The procedure carried out in the Compute step is identical for all robots.

In most papers in this area (cf. [9,13,25,26]), the robots are assumed to be rather limited. To begin with, the robots are assumed to have no means of directly communicating with each other. Moreover, they are assumed to be *oblivious* (or memoryless), namely, they cannot remember their previous states, their previous actions or the previous positions of the other robots. Hence the algorithm used in the Compute step cannot rely on information from previous cycles, and its only input is the current configuration. While this is admittedly an over-restrictive and unrealistic assumption, developing algorithms for the oblivious



model still makes sense in various settings, for two reasons. First, solutions that rely on non-obliviousness do not necessarily work in a dynamic environment where the robots are activated in different cycles, or robots might be added to or removed from the system dynamically. Secondly, any algorithm that works correctly for oblivious robots is inherently self-stabilizing, i.e., it withstands transient errors that alter the robots' local states into other (possibly illegal) states.

We consider the *fully asynchronous* timing model (cf. [8,9]). In this model, robots operate on their own (time-varying) rates, and no assumptions are made regarding the relative speeds of different robots. In particular, robots may remain inactive for arbitrarily long periods between consecutive operation cycles (subject to some “fairness” assumption that ensures that each robot is activated infinitely often in an infinite execution).

In order to describe the center of gravity algorithm, hereafter named Algorithm `Go.to_COG`, we use the following notation. Denote by  $\bar{r}_i[t]$  the location of robot  $i$  at time  $t$ . Denote the true center of gravity at time  $t$  by  $\bar{c}[t] = \frac{1}{N} \sum_{i=1}^N \bar{r}_i[t]$ . Denote by  $\bar{c}_i[t]$  the center of gravity as last calculated by the robot  $i$  before or at time  $t$ , i.e., if the last calculation by  $i$  was done at time  $t' \leq t$  then  $\bar{c}_i[t] = \bar{c}[t']$ . Note that, as mentioned before, robot  $i$  calculates this location in its own private coordinate system; however, for the purpose of describing the algorithm and its analysis, it is convenient to represent these locations in a unified global coordinate system (which of course is unknown to the robots themselves). This is justified by the linearity of the center of gravity calculation, which renders it invariant under any linear transformation. By convention  $\bar{c}_i[0] = \bar{r}_i[0]$  for all  $i$ .

Algorithm `Go.to_COG` is very simple. After measuring the current configuration at some time  $t$ , the robot  $i$  computes the average location of all robot positions (including its own),  $\bar{c}_i[t] = \sum_j \bar{r}_j[t]/N$ , and then proceeds to move towards the calculated point  $\bar{c}_i[t]$ . (As mentioned earlier, the move may terminate before the robot actually reaches the point  $\bar{c}_i[t]$ , but in case the robot has not reached  $\bar{c}_i[t]$ , it must have traversed a distance of at least  $S$ .)

## 2 Asynchronous Convergence

This section proves our main result, namely, that Algorithm `Go.to_COG` guarantees the convergence of  $N$  robots for any  $N \geq 2$  in the asynchronous model.

As noted in [10], the convex hull of the robot locations and calculated centers of gravity cannot increase in time. Intuitively, this is because (1) while a robot  $i$  performs a Move operation starting at time  $t_0$ ,  $\bar{c}_i$  does not change throughout the Move phase and  $\bar{r}_i[t]$  remains on the line segment  $[\bar{r}_i[t_0], \bar{c}_i[t_0]]$ , which is contained in the convex hull, and (2) when a robot performs a Look step, the calculated center of gravity is inside the convex hull of the  $N$  robot locations at that time. Hence we have the following.

**Lemma 1.** [10] *If for some time  $t_0$ ,  $\bar{r}_i[t_0]$  and  $\bar{c}_i[t_0]$  for all  $i$  reside in a closed convex curve,  $\mathcal{P}$ , then for every time  $t > t_0$ ,  $\bar{r}_i[t]$  and  $\bar{c}_i[t]$  also reside in  $\mathcal{P}$  for every  $1 \leq i \leq N$ .*



Hereafter we assume, for the time being, that the configuration at hand is one-dimensional, i.e., the robots reside on the  $x$ -axis. Later on, we extend the convergence proof to  $d$ -dimensions by applying the result to each dimension separately.

For every time  $t$ , let  $H[t]$  denote the convex hull of the points of interest  $\{\bar{r}_i[t] \mid 1 \leq i \leq N\} \cup \{\bar{c}_i[t] \mid 1 \leq i \leq N\}$ , namely, the smallest closed interval containing all  $2N$  points.

**Corollary 1.** *For  $N \geq 2$  robots and times  $t_1, t_0$ , if  $t_1 > t_0$  then  $H[t_1] \subseteq H[t_0]$ .*

Unfortunately, it is hard to prove convergence on the basis of  $h$  alone, since it is hard to show that  $h$  strictly decreases. Other potentially promising measures, such as  $\phi_1$  and  $\phi_2$  defined next, also prove problematic as they might sometimes increase in certain scenarios. Subsequently, the measure  $\psi$  we use in what follows to prove strict convergence is defined as a combination of a number of different measures. Formally, let us define the following quantities.

$$\begin{aligned}\phi_1[t] &= \sum_{i=1}^N |\bar{c}[t] - \bar{c}_i[t]|, \\ \phi_2[t] &= \sum_{i=1}^N |\bar{c}_i[t] - \bar{r}_i[t]|, \\ \phi[t] &= \phi_1[t] + \phi_2[t], \\ h[t] &= |H[t]|, \\ \psi[t] &= \frac{\phi[t]}{2N} + h[t].\end{aligned}$$

We now claim that  $\phi$ ,  $h$  and  $\psi$  are nonincreasing functions of time.

**Lemma 2.** *For every  $t_1 > t_0$ ,  $\phi[t_1] \leq \phi[t_0]$ .*

**Proof:** Examine the change in  $\phi$  due to the various robot actions. If a Look operation is performed by robot  $i$  at time  $t$ , then  $\bar{c}[t] - \bar{c}_i[t] = 0$  and  $|\bar{r}_i[t] - \bar{c}_i[t]| = |\bar{c}_i[t^*] - \bar{c}[t]|$  for any  $t^* \in [t', t]$ , where  $t'$  is the end of the last Move performed by robot  $i$ . Therefore,  $\phi$  is unchanged by the Look performed.

Now consider some time interval  $[t'_0, t'_1] \subseteq [t_0, t_1]$ , such that no Look operations were performed during  $[t'_0, t'_1]$ . Suppose that during this interval each robot  $i$  moved a distance  $\Delta_i$  (where some of these distances may be 0). Then  $\phi_2$  decreased by  $\sum_i \Delta_i$ , the maximum change in the center of gravity is  $|\bar{c}[t_1] - \bar{c}[t_0]| \leq \sum_i \Delta_i / N$ , and the robots' calculated centers of gravity have not changed. Therefore, the change in  $\phi_1$  is at most  $\phi_1[t_1] - \phi_1[t_0] \leq \sum_i \Delta_i$ . Hence, the sum  $\phi = \phi_1 + \phi_2$  cannot increase. ■

By Lemma 2 and Cor. 1, respectively,  $\phi$  and  $h$  are nonincreasing. Hence we have:

**Lemma 3.**  *$\psi$  is a nonincreasing function of time.*

**Lemma 4.** *For all  $t$ ,  $h \leq \psi \leq 2h$ .*

**Proof:** The lower bound is trivial. For the upper bound, notice that  $\phi$  is the sum of  $2N$  summands, each of which is at most  $h$  (since they all reside in the segment). ■

We now state a lemma which allows the analysis of the change in  $\phi$  (and therefore also  $\psi$ ) in terms of the contributions of individual robots.

**Lemma 5.** *If by the action of a robot  $i$  separately, in the time interval  $[t_0, t_1]$  its contribution to  $\phi$  is  $\delta_i$ , then  $\phi[t_1] \leq \phi[t_0] + \delta_i$ .*

**Proof:** Lemma 2 implies that Look actions have no effect on  $\phi$  and therefore can be disregarded. A robot moving a distance  $\Delta_i$  will always decrease its term in  $\phi_2$  by  $\Delta_i$ , and the motions of other robots have no effect on this term. Denote by  $\Delta_j$  the motions of the other robots. Notice that

$$|\bar{c} + \frac{\Delta_i}{N} + \frac{1}{N} \sum_{j \neq i} \Delta_j - \bar{c}_k| \leq |\bar{c} + \frac{\Delta_i}{N} - \bar{c}_k| + \frac{1}{N} \sum_{j \neq i} |\Delta_j|.$$

The function  $\phi_1$  contains  $N$  summands, each of which contains a contribution of at most  $\frac{1}{N} |\Delta_j|$  from every robot  $j \neq i$ . Therefore, the total contribution of each robot to  $\phi_1$  is at most  $|\Delta_j|$ , which is canceled by the negative contribution of  $|\Delta_j|$  to  $\phi_2$ . ■

**Theorem 1.** *For every time  $t_0$ , there exists some time  $\hat{t} > t_0$  such that*

$$\psi[\hat{t}] \leq \left(1 - \frac{1}{8N^2}\right) \psi[t_0].$$

**Proof:** Assume without loss of generality that at time  $t_0$ , the robots and centers of gravity resided in the interval  $H[t_0] = [0, 1]$  (and thus  $h[t_0] = 1$  and  $\psi[t_0] \leq 2$ ). Take  $t^*$  to be the time after  $t_0$  when each robot has completed at least one entire Look–Compute–Move cycle. There are now two possibilities:

1. Every center of gravity  $\bar{c}_i[t']$  that was calculated at time  $t' \in [t, t^*]$  resided in the segment  $(\frac{1}{2N}, 1]$ . In this case at time  $t^*$  no robot can reside in the segment  $[0, \frac{1}{2N}]$  (since every robot has completed at least one cycle operation, where it has arrived at its calculated center of gravity outside the segment  $[0, \frac{1}{2N}]$ , and from then on it may have moved a few more times to its newly calculated centers of gravity, which were also outside this segment). Hence at time  $\hat{t} = t^*$  all robots and centers of gravity reside in  $H[\hat{t}] \subseteq [\frac{1}{2N}, 1]$ , so  $h[\hat{t}] \leq 1 - \frac{1}{2N}$ , and  $\phi[\hat{t}] \leq \phi[t_0]$ . Therefore,

$$\psi[\hat{t}] = \frac{\phi[\hat{t}]}{2N} + h[\hat{t}] \leq \frac{\phi[t_0]}{2N} + 1 - \frac{1}{2N} = \psi[t_0] - \frac{1}{2N}.$$

Also, by Lemma 4,  $\psi[t] \leq 2$ , hence  $\frac{1}{2N} \geq \frac{1}{4N} \psi[t_0]$ . Combined, we get  $\psi[\hat{t}] \leq (1 - \frac{1}{4N}) \psi[t_0]$ .

2. For some  $t_1 \in [t, t^*]$ , the center of gravity  $\bar{c}_i[t_1] = \frac{1}{N} \sum_{j=1}^N \bar{r}_j[t_1]$  calculated by some robot  $i$  at time  $t_1$  resided in  $[0, \frac{1}{2N}]$ . Therefore, at time  $t_1$  all robots resided in the segment  $[0, \frac{1}{2}]$  (by Markov inequality [11]). We split again into two subcases:

- a) At time  $t_1$  all centers of gravity,  $\bar{c}_i[t_1]$  resided in the interval  $[0, \frac{3}{4}]$ . In this case, take  $\hat{t} = t^*$ ,  $h[\hat{t}] \leq h[t_1] \leq \frac{3}{4} \leq \frac{3}{4}h[t_0]$ , and therefore

$$\psi[\hat{t}] = h[\hat{t}] + \frac{\phi[\hat{t}]}{2N} \leq \frac{3}{4}h[t_0] + \frac{\phi[t_0]}{2N} \leq \frac{7}{8}h[t_0] + \frac{7}{8} \frac{\phi[t_0]}{2N} = \frac{7}{8}\psi[t_0],$$

where the last inequality is due to the fact that  $\frac{\phi[t_0]}{2N} \leq h[t_0]$  as argued in the proof of Lemma 4. The theorem immediately follows.

- b) At time  $t_1$  there existed robots with  $\bar{c}_i[t_1] > \frac{3}{4}$ . In this case, take  $k$  to be the robot with the highest center of gravity (or one of them) and take  $\hat{t}$  to be the time robot  $k$  completes its next Move. Its move size is at least  $\Delta_k \geq \frac{1}{4}$ , hence its motion decreased  $|\bar{r}_k - \bar{c}_k|$  by  $\Delta_k \geq \frac{1}{4}$ , and  $\phi_2$  is decreased by  $\Delta_k$ . Also, The robots and calculated centers of gravity are now restricted to the segment  $[0, \bar{c}_k]$ . Since the true center of gravity must be to the left of  $\bar{c}_k$ ,  $\bar{c} - \bar{c}_k$  is decreased by  $\Delta_k/N$ . The sum of all other summands in  $\phi_1$  may increase by at most  $(N-1)\Delta_k/N$ . By Lemma 5  $\phi$  can be bounded from above by the contribution of a single robot. Therefore,  $\phi$  decreased by at least  $2\Delta_k/N$ , and the term  $\frac{\phi}{2N}$  in  $\psi$  decreased by at least  $\frac{2\Delta_k/N}{2N} = \frac{\Delta_k}{N^2} \geq \frac{1}{4N^2}$ . As  $\psi[t_0] \leq 2$  and  $h$  is nonincreasing,  $\psi$  decreased by at least  $\frac{1}{4N^2} \geq \frac{\psi[t_0]}{8N^2}$ . The theorem follows. ■

To prove the convergence of the gravitational algorithm in  $d$ -dimensional Euclidean space, apply Theorem 1 to each dimension separately. Observe that by Theorem 1 and Lemma 4, for every  $\epsilon > 0$  there is a time  $t_\epsilon$  by which  $h[t_\epsilon] \leq \psi[t_\epsilon] \leq \epsilon$ , hence the robots have converged to an  $\epsilon$ -neighborhood.

**Theorem 2.** *For any  $N \geq 2$ , in  $d$ -dimensional Euclidean space,  $N$  robots performing Algorithm `Go-to-COG` will converge.*

### 3 Convergence Rate

To bound the rate of convergence in the fully asynchronous model, one should make some normalizing assumption on the operational speed of the robots. A standard type of assumption is based on defining the maximum length of a robot cycle during the execution (i.e., the maximum time interval between two consecutive Look steps of the same robot) as one time unit. For our purposes it is more convenient to make the slightly modified assumption that for every time  $t$ , during the time interval  $[t, t+1]$  every robot has completed at least one cycle. Note that the two assumptions are equivalent up to a constant factor of 2. Note also that this assumption is used only for the purpose of complexity analysis, and was not used in our correctness proof.

**Lemma 6.** *For every time interval  $[t_0, t_1]$ ,*

$$\psi[t_1] \leq \left(1 - \frac{1}{8N^2}\right)^{\lfloor \frac{t_1 - t_0}{2} \rfloor} \psi[t_0].$$

**Proof:** Consider the different cases analyzed in the proof of Theorem 1. By our timing assumption, we can take  $t^* = t_0 + 1$ . In case 1,  $\psi$  is decreased by a factor of  $1 - \frac{1}{4N}$  by time  $\hat{t} = t^*$ , i.e., within one time unit. In case 2a,  $\psi$  is decreased by a factor of  $\frac{7}{8}$  by time  $\hat{t} = t^*$ , i.e., within one time unit again. The slowest convergence rate is obtained in case 2b. Here, we can take  $\hat{t} = t^* + 1 = t_0 + 2$ , and conclude that  $\psi$  is decreased by a factor of  $1 - \frac{1}{8N^2}$  after two time units. The lemma follows by assuming a worst case scenario in which during the time interval  $[t_0, t_1]$ , the “slow” case 2b is repeated for  $\lfloor \frac{t_1 - t_0}{2} \rfloor$  times. ■

By the two inequalities of Lemma 4 we have that  $h[t_1] \leq \psi[t_1]$  and  $\psi[t_0] \leq 2h[t_0]$ , respectively. Lemma 6 now yields the following.

**Theorem 3.** *For every time interval  $[t_0, t_1]$ ,*

$$h[t_1] \leq 2 \left(1 - \frac{1}{8N^2}\right)^{\lfloor \frac{t_1 - t_0}{2} \rfloor} h[t_0].$$

**Corollary 2.** *In any execution of the gravitational algorithm in the asynchronous model, over every interval of  $O(N^2)$  time units, the size of the  $d$ -dimensional convex hull of the robot locations and centers of gravity is halved in each dimension separately.* ■

An example for slow convergence is given by the following lemma.

**Lemma 7.** *There exist executions of the gravitational algorithm in which  $\Omega(N)$  time is required to halve the convex hull of  $N$  robots in each dimension.*

**Proof:** Initially, and throughout the execution, the  $N$  robots are organized on the  $x$  axis. The execution consists of phases of the following structure. Each phase takes exactly one time unit, from time  $t$  to time  $t + 1$ . At each (integral) time  $t$ , robot 1 is at one endpoint of the bounding segment  $H[t]$  while the other  $N - 1$  robots are at the other endpoint of the segment. Robot 1 performs a Look at time  $t$  and determines its perceived center of gravity  $\bar{c}_i[t]$  to reside at a distance  $\frac{h[t]}{N-1}$  from the distant endpoint. Next, the other  $N - 1$  robots perform a long sequence of (fast) cycles, bringing them to within a distance  $\epsilon$  of robot 1, for arbitrarily small  $\epsilon$ . Robot 1 then performs its movement to its perceived center of gravity  $\bar{c}_i[t]$ . Hence the decrease in the size of the bounding interval during the phase is  $h[t] - h[t + 1] = \frac{h[t]}{N-1} + \epsilon$ , or in other words, at the end of the phase  $h[t + 1] \approx \left(1 - \frac{1}{N-1}\right) h[t]$ . It follows that  $O(N)$  steps are needed to reduce the interval size to a  $1/e$  fraction of its original size. ■

Note that there is still a linear gap between the upper and lower bounds on the convergence rate of the gravitational algorithm as stated in Cor. 2 and Lemma 7.

It is interesting to compare these bounds with what happens in a variant of the fully synchronous model (cf. [27]), in which all robots operate at fixed time cycles, and the Look phase of all robots is simultaneous. It is usually assumed that the robots do not necessarily reach their desired destination. However, there is some constant minimum distance,  $S$ , which the robots are guaranteed to traverse at every step. Therefore, if  $H[0]$  is the convex hull of the  $N$  robots at time 0 and  $h[0]$  is the maximum width of  $H[0]$  in any of the  $d$  dimensions, then we have the following.

**Lemma 8.** *In any execution of the gravitational algorithm in the fully synchronous model, the robots achieve gathering in at most  $\lceil 4h[0]d^{3/2}/S \rceil$  time.*

**Proof:** If the distance of each robot from the center of gravity is at most  $S$ , then at the next step they will all gather. Suppose now that there exists at least one robot whose distance from the center of gravity is greater than  $S$ . Since the center of gravity is within the convex hull, the largest dimension is at least  $h[0] \geq S/\sqrt{d}$ . Without loss of generality, assume that the projection of the hull on the maximum width dimension is on the interval  $[0, a]$ , and that the projection of the center of gravity  $\bar{c}[0]$  is in the interval  $[\frac{a}{2}, a]$ . Then in each step, every robot moves by a vector  $\min\{\bar{r}_i - \bar{c}, S' \frac{\bar{r}_i - \bar{c}}{|\bar{r}_i - \bar{c}|}\}$  for some  $S' \geq S$ . By assumption,  $a$  is the width of the largest dimension and therefore  $a \geq |\bar{r}_i - \bar{c}|/\sqrt{d}$ . For every robot in the interval  $[0, \frac{a}{4}]$ , the distance to the current center of gravity will decrease in the next step by at least  $\min\{\frac{a}{4}, S \frac{a/4}{a\sqrt{d}}\} \geq \frac{S}{4\sqrt{d}}$ . Thus, the width of at least one dimension decreases by at least  $\frac{S}{4\sqrt{d}}$  in each step. Therefore, gathering is achieved after at most  $\lceil 4h[0]d^{3/2}/S \rceil$  cycles, independently of  $N$ . ■

## 4 Fault Tolerance

In this section we consider the behavior of the gravitational algorithm in the presence of possible robot failures.

Let us first consider a model allowing only *transient* failures. Such a failure causes a change in the states of some robots, possibly into illegal states. Notice that Theorem 2 makes no assumptions about the initial positions and centers of gravity, other than that they are restricted to some finite region. It follows that, due to the oblivious nature of the robots (and hence the algorithm), the robots will converge regardless of any finite number of transient errors occurring in the course of the execution.

We now turn to consider the *crash* fault model. This model, presented in [1], follows the common crash (or “fail-stop”) fault model in distributed computing, and assumes that a robot may fail by halting. This may happen at any point in time during the robot’s cycle, i.e., either during the movement towards the goal point or before it has started. Once a robot has crashed, it will remain stationary indefinitely.

In [1], it is shown that in the presence of a single crash fault, it is possible to *gather* the remaining (functioning) robots to a common point. Here, we avoid the gathering requirement and settle for the weaker goal of convergence. We show that the `Go_to_COG` algorithm converges for every number of crashed robots. In fact, in a sense convergence is easier in this setting since the crashed robots determine the final convergence point for the nonfaulty robots. We have the following.

**Theorem 4.** *Consider a group of  $N$  robots that execute Algorithm `Go_to_COG`. If  $1 \leq M \leq N - 2$  robots crash during the execution, then the remaining  $N - M$  robots will converge to the center of gravity of the crashed robots.*

**Proof:** Consider an execution of the gravitational algorithm by a group of  $N$  robots. Without loss of generality assume that the crashed robots were  $1, \dots, M$ , and their crashing times were  $t_1 \leq \dots \leq t_M$ , respectively. Consider the behavior of the algorithm starting from time  $t_M$ . For the analysis, a setting in which the  $M$  robots crashed at general positions  $\bar{r}_1, \dots, \bar{r}_M$  is equivalent to one in which all  $M$  crashed robots are concentrated in the center of gravity  $\frac{1}{M} \sum_{i=1}^M \bar{r}_i$ . Assume, without loss of generality, that this center of gravity is at 0.

Now consider some time  $t_0 \geq t_M$ . Let  $H[t_0] = [a, b]$  for some  $a \leq 0 \leq b$ . By Corollary 1, the robots will remain in the segment  $[a, b]$  at all times  $t \geq t_0$ . The center of gravity calculated by any nonfaulty robot  $M + 1 \leq j \leq N$  at time  $t \geq t_0$  will then be

$$\bar{c}_j[t] = \frac{1}{N} \sum_{i=1}^N \bar{r}_i = \frac{1}{N} \left( M \cdot 0 + \sum_{i=M+1}^N \bar{r}_i[t] \right).$$

Hence, all centers of gravity calculated hereafter will be restricted to the segment  $[a', b']$  where  $a' = \frac{N-M}{N} \cdot a$  and  $b' = \frac{N-M}{N} \cdot b$ . Consequently, denoting by  $\hat{t}$  the time by which every nonfaulty robot has completed a Look-Compute-Move cycle, we have that  $H[\hat{t}] \subseteq [a', b']$ , hence  $h[\hat{t}] \leq \frac{N-M}{N} \cdot h[t_0]$ . Again, the argument can be extended to any number of dimensions by considering each dimension separately. It follows that the robots converge to a point. ■

## References

1. N. Agmon and D. Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms*, pages 1063–1071, January 2004.
2. H. Ando, Y. Oasa, I. Suzuki, and M. Yamashita. A distributed memoryless point convergence algorithm for mobile robots with limited visibility. *IEEE Trans. Robotics and Automation*, 15:818–828, 1999.
3. H. Ando, I. Suzuki, and M. Yamashita. Formation and agreement problems for synchronous mobile robots with limited visibility. In *Proc. IEEE Symp. of Intelligent Control*, pages 453–460, August 1995.
4. T. Balch and R. Arkin. Behavior-based formation control for multi-robot teams. *IEEE Trans. on Robotics and Automation*, 14, December 1998.

5. G. Beni and S. Hackwood. Coherent swarm motion under distributed control. In *Proc. DARS'92*, pages 39–52, 1992.
6. Y.U. Cao, A.S. Fukunaga, and A.B. Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4(1):7–23, March 1997.
7. Y.U. Cao, A.S. Fukunaga, A.B. Kahng, and F. Meng. Cooperative mobile robots: Antecedents and directions. In *Proc. Int. Conf. of Intel. Robots and Sys.*, pages 226–234, 1995.
8. M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro. Solving the robots gathering problem. In *Proc. 30th Int. Colloq. on Automata, Languages and Programming*, pages 1181–1196, 2003.
9. M. Cieliebak and G. Prencipe. Gathering autonomous mobile robots. In *Proc. 9th Int. Colloq. on Structural Information and Communication Complexity*, pages 57–72, June 2002.
10. R. Cohen and D. Peleg. Robot convergence via center-of-gravity algorithms. In *Proc. 11th Colloq. on Structural Information and Communication Complexity*, 2004. To appear.
11. W. Feller. *An introduction to Probability Theory and its Applications*. Wiley, New York, 1968.
12. P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Hard tasks for weak robots: The role of common knowledge in pattern formation by autonomous mobile robots. In *Proc. 10th Int. Symp. on Algorithms and Computation*, pages 93–102, 1999.
13. P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Gathering of autonomous mobile robots with limited visibility. In *Proc. 18th Symp. on Theoretical Aspects of Computer Science*, pages 247–258, February 2001.
14. D. Jung, G. Cheng, and A. Zelinsky. Experiments in realising cooperation between autonomous mobile robots. In *Proc. Int. Symp. on Experimental Robotics*, June 1997.
15. Y. Kawauchi, M. Inaba, and T. Fukuda. A principle of decision making of cellular robotic system (CEBOT). In *Proc. IEEE Conf. on Robotics and Automation*, pages 833–838, 1993.
16. M.J. Mataric. *Interaction and Intelligent Behavior*. PhD thesis, MIT, 1994.
17. S. Murata, H. Kurokawa, and S. Kokaji. Self-assembling machine. In *Proc. IEEE Conf. on Robotics and Automation*, pages 441–448, 1994.
18. L.E. Parker. Designing control laws for cooperative agent teams. In *Proc. IEEE Conf. on Robotics and Automation*, pages 582–587, 1993.
19. L.E. Parker. On the design of behavior-based multi-robot teams. *J. of Advanced Robotics*, 10, 1996.
20. L.E. Parker and C. Touzet. Multi-robot learning in a cooperative observation task. In *Distributed Autonomous Robotic Systems 4*, pages 391–401, 2000.
21. L.E. Parker, C. Touzet, and F. Fernandez. Techniques for learning in multi-robot teams. In T. Balch and L.E. Parker, editors, *Robot Teams: From Diversity to Polymorphism*. A. K. Peters, 2001.
22. G. Prencipe. CORDA: Distributed coordination of a set of autonomous mobile robots. In *Proc. 4th European Research Seminar on Advances in Distributed Systems*, pages 185–190, May 2001.
23. G. Prencipe. *Distributed Coordination of a Set of Autonomous Mobile Robots*. PhD thesis, Università Degli Studi Di Pisa, 2002.
24. K. Sugihara and I. Suzuki. Distributed algorithms for formation of geometric patterns with many mobile robots. *Journal of Robotic Systems*, 13(3):127–139, 1996.

25. I. Suzuki and M. Yamashita. Agreement on a common x-y coordinate system by a group of mobile robots. In *Proc. Dagstuhl Seminar on Modeling and Planning for Sensor-Based Intelligent Robots*, September 1996.
26. I. Suzuki and M. Yamashita. Distributed anonymous mobile robots - formation and agreement problems. In *Proc. 3rd Colloq. on Structural Information and Communication Complexity*, pages 313–330, 1996.
27. I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM J. on Computing*, 28:1347–1363, 1999.
28. I.A. Wagner and A.M. Bruckstein. From ants to a(ge)nts. *Annals of Mathematics and Artificial Intelligence*, 31, special issue on ant-robotics:1–5, 1996.



# The Average Case Analysis of Partition Sorts<sup>\*</sup>

Richard Cole and David C. Kandathil

Computer Science Department, Courant Institute, New York University,  
251 Mercer Street, New York, NY 10012, USA.

**Abstract.** This paper introduces a new family of in-place sorting algorithms, the *partition sorts*. They are appealing both for their relative simplicity and their efficient performance. They perform  $\Theta(n \log n)$  operations on the average, and  $\Theta(n \log^2 n)$  operations in the worst case.

The partition sorts are related to another family of sorting algorithms discovered recently by Chen [Che02]. He showed empirically that one version ran faster, on the average, than quicksort, and that the algorithm family performed  $\Theta(n \log n)$  comparisons in the worst case; however no average case analysis was obtained.

This paper completes the analysis of Chen's algorithm family. In particular, a bound of  $n \log n + O(n)$  comparisons and  $\Theta(n \log n)$  operations is shown for the average case, and  $\Theta(n \log^2 n)$  operations for the worst case. The average case analysis is somewhat unusual. It proceeds by showing that Chen's sorts perform, on the average, no more comparisons than the partition sorts.

Optimised versions of the partition sort and Chen's algorithm are very similar in performance, and both run marginally faster than an optimised quasi-best-of-nine variant of quicksort [BM93]. They both have a markedly smaller variance than the quicksorts.

## 1 Introduction

We consider the problem of in-place sorting of internally stored data with no a priori structure. We focus on the average case performance of deterministic sequential sorting algorithms. By in-place, we mean algorithms which use an additional  $O(\log n)$  space; by average case, that all input orderings are equally likely.

A new family of in-place algorithms, which have a quicksort-like flavour, was proposed by Chen [Che02], and their worst case number of comparisons was shown to be  $\Theta(n \log n)$ . Empirical evidence was presented for the conjecture that their average case number of comparisons is close to the information theoretic lower bound and the fact that some versions run faster than quicksort on the average (for medium sized sets of order  $10^4$ ). The average case analysis of comparisons was posed as an open problem. Bounds on operation counts were not mentioned.

---

<sup>\*</sup> This work was supported in part by NSF grant CCR0105678.

In this paper, we introduce a related new family of in-place algorithms, which we call *partition sorts*. The partition sorts are attractive, both because of their relative simplicity, and their asymptotic and practical efficiency. We show the following performance bounds for partition sort:

1. The average case number of comparisons is  $n \log n + O(n)$ , and for one version, is at most  $n \log(n+1) - 1.193n + O(\log n)$  comparisons<sup>1</sup>. They perform  $\Theta(n \log^2 n)$  comparisons in the worst case.
2. They perform  $\Theta(n \log n)$  operations on the average and  $\Theta(n \log^2 n)$  operations in the worst case.

As we show by means of a non-trivial argument, the average comparison cost of the partition sorts upper bounds that of Chen's algorithm family; this bound is also used in the analysis of exchanges for Chen's sorts. Using this and additional arguments, we obtain the following bounds for Chen's family of algorithms:

1. The number of comparisons performed by each version of Chen's algorithm is no more than the number of comparisons performed by a corresponding version of partition sort, on the average. Hence Chen's algorithm family performs  $n \log n + O(n)$  comparisons on the average. We also give a simple and tight analysis for the worst case number of comparisons.
2. They perform  $\Theta(n \log n)$  exchanges on the average and  $\Theta(n \log^2 n)$  exchanges in the worst case.

The quicksort partition procedure is a central routine in both families of algorithms, and as a consequence, they both have excellent caching behaviour. Their further speedup relative to quicksort is due to a smaller expected depth of recursion.

We outline the analysis of the partition sort (Section 2) and the average case cost of comparisons for Chen's algorithm (Section 3). We also present some empirical studies of these algorithms (Section 4).

## Prior Work

It is well known [FJ59] that, for comparison based sorts of  $n$  items,  $\log n! = n \log n - n \log e + \frac{1}{2} \log n + O(1) = n \log n - 1.443n + O(\log n)$  is a lower bound on both the average and the worst case number of comparisons, in the decision tree model. Merge-insertion sort [FJ59] has the best currently known worst case comparison count:  $n \log(\frac{3}{4}n) - n + O(\log n) = n \log n - 1.441n + O(\log n)$ , at the expense of  $\Theta(n^2)$  exchanges<sup>2</sup>; it is, however, not clear how to implement it in  $\Theta(n \log n)$  operations while maintaining this comparison count; the best result is an in-place mergesort which uses merge-insertion sort for constant size

<sup>1</sup> We let  $\log n$  denote the *piecewise log function*  $\lfloor \log n \rfloor + \frac{2}{n}(n - 2^{\lfloor \log n \rfloor})$ , which is an approximation to the log function, matching it at  $n = 2^k$ , for integers  $k \geq 0$ .

<sup>2</sup>  $\log n \stackrel{\text{def}}{=} \lfloor \log \frac{3}{4}n \rfloor + \log \frac{4}{3} + \frac{2}{n}(n - \frac{4}{3}2^{\lfloor \log \frac{3}{4}n \rfloor})$ , i.e., the approximation to the log function with equality roughly when  $n = \frac{4}{3}2^k$ , for integers  $k \geq 0$ .

subproblems [Rei92]; this achieves  $\Theta(n \log n)$  operations and comes arbitrarily close to the above comparison count, at the cost of increasingly large constants in the operation count. The trivial lower bound of  $\Theta(n)$  on both the average and the worst case number of exchanges is met by selection sort at the expense of  $\Theta(n^2)$  comparisons. A different direction is to simultaneously achieve  $\Theta(n \log n)$  comparisons,  $\Theta(n)$  exchanges, and  $\Theta(1)$  additional space, thus meeting the lower bounds on all resources; this was recently attained [FG03] but its efficiency in practice is unclear.

Hoare's quicksort [Hoa61, Hoa62] has been the in-place sorting algorithm with the best currently known average case performance, namely  $2n \ln n = 1.386n \log n$  comparisons, with worst case number of exchanges  $\Theta(n \log n)$ ; it runs in  $\Theta(\log n)$  additional space with modest exchange counts, and has excellent caching behaviour. Its worst case number of comparisons, however, is  $\Theta(n^2)$ . These facts remain essentially the same for the many deterministic variants of the algorithm that have been proposed (such as the best-of-three variant [Hoa62, Sin69] which performs  $\frac{12}{7}n \ln n = 1.188n \log n$  comparisons on the average, and the quasi-best-of-nine variant [BM93] which empirically seems to perform  $1.094n \log n$  comparisons on the average).

Classical mergesort performs  $n \log n - n + 1$  comparisons and  $\Theta(n \log n)$  operations in the worst case and exhibits good caching behaviour, but is not in-place, requiring  $\Theta(n)$  additional space. In-place mergesorts with  $\Theta(1)$  additional space have been achieved [Rei92, KPT96] with the same bounds but their complex index manipulations slow them down in practice.

Heapsort, due to Williams and Floyd [Wil64, Flo64], is the only practical in-place sorting algorithm known that performs  $\Theta(n \log n)$  operations in the worst case. Bottom-up heapsort [Weg93], originally due to Floyd, performs  $n \log n + O(n)$  comparisons on the average and  $\frac{3}{2}n \log n + O(n)$  comparisons in the worst case, with  $\Theta(1)$  additional space; weak-heapsort [Dut93, EW00] performs  $n \log n + 1.1n$  comparisons in the worst case while using  $n$  additional bits. The exchange counts are also modest. Nonetheless, its average case behaviour is not competitive with that of quicksort; it does not exhibit substantial locality of reference and deteriorates due to caching effects [LL97].

## 2 The Partition Sort

We present the partition sort as a one parameter family of algorithms, with parameter  $\gamma > 1$ .

The sort of an array of  $n$  items begins with a recursive sort of the first  $\left\lfloor \frac{n}{\gamma} \right\rfloor$  items, forming a subarray  $S$  of sorted items.

The heart of the algorithm is a *partition sort completion procedure* that completes the sort of the following type of partially sorted array. The array consists of two adjacent subarrays  $S$  and  $U$ : The portion to the left,  $S$ , is sorted; the remainder,  $U$ , is unsorted;  $s \stackrel{\text{def}}{=} |S|$  and  $u \stackrel{\text{def}}{=} |U|$ . For a call  $(S, U)$ , sort completion proceeds in two steps:

*Multiway Partitioning:* The items of  $U$  are partitioned into  $s + 1$  buckets,  $U_0, U_1, \dots, U_s$ , defined by consecutive items of the sorted subarray  $S = (s_1, s_2, \dots, s_s)$ , where  $s_1 < s_2 < \dots < s_s$ .  $U_k$  contains the items in  $U$  lying between  $s_k$  and  $s_{k+1}$  (we define  $s_0 \stackrel{\text{def}}{=} -\infty$  and  $s_{s+1} \stackrel{\text{def}}{=} +\infty$ ). Thus the items of  $S$  act as pivots in a multiway partitioning of  $U$ . We describe the implementation of this partitioning below.

*Sorting of Buckets:* Each bucket  $U_k$ ,  $0 \leq k \leq s$ , is sorted by insertion sort. In order to bound the worst case operation count, if  $|U_k| > c \log n$  (where  $c$  is a suitable constant)  $U_k$  is sorted recursively; in this case we say  $U_k$  is *large*, otherwise we say it is *small*.

The multiway partitioning may be thought of as performing, for each item in  $U$ , a binary search over the sorted subarray  $S$ . Nevertheless, so as to implement it in-place and to minimise the number of exchanges performed, we use a recursive multiway partitioning procedure, essentially due to Chen [Che02], described below for a call  $(S, U)$ .

First the unsorted subarray  $U$  is partitioned about the median  $x$  of  $S$ ; this creates the ordering  $S_L x S_R U_L U_R$ , with  $S_L < x < S_R$  and  $U_L < x < U_R$ <sup>3</sup>. Next the blocks  $\{x\} \cup S_R$  and  $U_L$  are swapped (as described in the next paragraph), preserving the order of  $\{x\} \cup S_R$  but not necessarily of  $U_L$ , yielding the ordering  $S_L U'_L x S_R U_R$  with  $S_L \cup U'_L < x < S_R \cup U_R$ . Then the subarrays  $(S_L, U'_L)$  and  $(S_R, U_R)$  are partitioned recursively. The base case for the recursion arises when the sorted subarray  $S$  is empty; then the unsorted subarray  $U$  forms one of the sought buckets.

The partitioning of  $U$  about  $x$  may be done using any of the partitioning routines developed for quicksort. A simple way of swapping blocks  $\{x\} \cup S_R$  and  $U_L$  is to walk through items in  $\{x\} \cup S_R$  from right to left, swapping each item  $a$  thus encountered with the item in  $a$ 's destination position.

It is readily seen that each item  $u \in U$  performs exactly the same comparisons as in a binary search.

By solving the smaller subproblem first, and eliminating the remaining tail recursion, the additional space needed may be limited to  $\Theta(\log n)$  in the worst case.

*Remark 1.* The case  $\gamma = \sqrt{n}$  corresponds to an in-place sequential implementation of parallel quicksort [Rei85].

## 2.1 Worst Case Analysis of Operations

We analyse the algorithm for the case  $\gamma = 2$  in detail, focusing on the comparisons initially.

Observe that the number of comparisons occurring during the insertion sorts is bounded by  $O(n \log n)$ . Thus, it suffices to analyse the remaining comparisons.

<sup>3</sup>  $a < B$  means that for all  $b \in B$ ,  $a < b$ .  $B < c$  is defined analogously.

Let  $B(s, u)$  be the worst case number of non-insertion sort comparisons for sort completions on inputs of size  $(s, u)$ , and let  $C(n)$  be the worst case number of non-insertion sort comparisons for the partition sort on inputs of size  $n$ .

Each item  $u \in U$  performs at most  $\lceil \log(s+1) \rceil$  comparisons during the multiway partitioning. It remains to bound the comparisons occurring during the sorting of buckets. If  $l_k \stackrel{\text{def}}{=} |U_k|$ , then  $\sum_{k=0}^s l_k = u$ , and

$$B(s, u) \leq \max_{l_k} \left( u \lceil \log(s+1) \rceil + \sum_{k=0}^s C(l_k) \right) = u \lceil \log(s+1) \rceil + C(u)$$

where we may have overestimated since a term  $C(l_k)$  is needed only if  $l_k \geq c \log n$ .

Clearly,  $C(1) = 0$  and, for  $n > 1$ :

$$C(n) \leq B\left(\left\lfloor \frac{n}{2} \right\rfloor, \left\lceil \frac{n}{2} \right\rceil\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \leq \left\lceil \frac{n}{2} \right\rceil \left\lceil \log\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \right\rceil + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

Thus,  $C(n) = \Theta(n \log^2 n)$ , since the bound is tight on an input in sorted order, and we have shown:

**Lemma 1.** *The partition sort, with  $\gamma = 2$ , performs  $\Theta(n \log^2 n)$  comparisons in the worst case.*

For larger  $\gamma$ , including the operation count, we can show:

**Theorem 1.** *The partition sort performs  $\Theta(\gamma n \log^{\frac{2}{\gamma}} n)$  operations in the worst case.*

## 2.2 Average Case Analysis of Operations

Again, we analyse the case  $\gamma = 2$  in detail; as before, the comparison count dominates the operation count, and thus it suffices to analyse comparisons.

For simplicity, we assume initially that the algorithm sorts all buckets large and small using insertion sort. We will ensure, with a suitable choice of  $c$ , that precisely because we forsake recursive calls to partition sort, this introduces an inefficiency in the sorting of large buckets on the average, and hence that the average case bound that we will obtain is a valid (if slightly weak) bound for the original algorithm too.

Let  $B(s, u)$  be the average case number of comparisons for sort completions on inputs of size  $(s, u)$ , and  $C(n)$ , the average case number of comparisons for the partition sort on inputs of size  $n$ . Let  $I(l)$  denote the average case number of comparisons for sorting a bucket of size  $l$ . It is easy to show:

**Lemma 2.** *If  $s+1 = 2^i + h$  with  $0 \leq h < 2^i$ , then the partitioning for an input of size  $(s, u)$  performs  $u \left(i + \frac{2h}{s+1}\right)$  comparisons on the average.*

We now bound  $B(s, u)$  by overestimating the number of comparisons required, on the average, for the sorting of buckets; with  $s + 1 = 2^i + h$ , where  $0 \leq h < 2^i$ , we have:

$$B(s, u) = u \left( i + \frac{2h}{s+1} \right) + \sum_{\text{item } a} \sum_{l=2}^u \Pr[a \text{ is in } U \text{ and its bucket has size } l] \frac{I(l)}{l}$$

The probability for an arbitrary item  $a$  in  $S \cup U$  to be in  $U$  and to be in a bucket of size  $l$ , may be overestimated as follows. For an arbitrary item  $a$ , the end points of a bucket of size  $l$  (which are in  $S$ ) may be chosen in at most  $l$  distinct ways, and given such a pair of end points, the probability that they form a bucket (which is the same as saying that the remaining  $s - 2$  items in  $S$  are chosen from outside this range) is exactly:

$$\binom{s+u-l-2}{s-2} \bigg/ \binom{s+u}{s}$$

unless  $a$  is among the least or greatest  $l$  items of  $S \cup U$ ; to avoid anomalies, it suffices to pretend the buckets at the two extremes are combined to form a single bucket; then the same probability applies to every item, and clearly we have only overestimated the cost of the algorithm. Hence:

$$B(s, u) \leq u \left( i + \frac{2h}{s+1} \right) + (s+u) \sum_{l=2}^u I(l) \binom{s+u-l-2}{s-2} \bigg/ \binom{s+u}{s}$$

We may now obtain a bound on  $C(n)$ . For  $n > 1$ , we have:

$$\begin{aligned} C(n) &\leq B\left(\left\lfloor \frac{n}{2} \right\rfloor, \left\lceil \frac{n}{2} \right\rceil\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\ &\leq \left\lceil \frac{n}{2} \right\rceil \left( i + \frac{2h}{\left\lfloor \frac{n}{2} \right\rfloor + 1} \right) + n \left[ \sum_{l=2}^{\left\lceil \frac{n}{2} \right\rceil} \frac{\binom{n-l-2}{\left\lfloor \frac{n}{2} \right\rfloor - 2}}{\binom{n}{\left\lfloor \frac{n}{2} \right\rfloor}} I(l) \right] + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \end{aligned}$$

where  $\left\lfloor \frac{n}{2} \right\rfloor + 1 = 2^i + h$  and  $0 \leq h < 2^i$ .

Clearly,  $I(l) \leq \binom{l}{2}$ .

**Lemma 3.**

$$\sum_{l=2}^{\frac{n+1}{2}} \frac{\binom{n-l-2}{\frac{n-5}{2}}}{\binom{n}{\frac{n-1}{2}}} \binom{l}{2} \leq \frac{1}{2} + \frac{12}{n} + O\left(\frac{1}{n^2}\right)$$

Substitutiong in the equation for  $C(n)$  yields:

**Lemma 4.** *If all input orderings are equally likely, the partition sort, with  $\gamma = 2$ , performs at most  $n \log(n+1) - n + O(\log n)$  comparisons on the average.*

By bounding  $I(l)$  more exactly, we can show:

**Theorem 2.** *If all input orderings are equally likely, the partition sort, with  $\gamma = 2$ , performs at most  $n \log(n+1) - \left(\frac{1}{2} + \ln 2\right)n + O(\log n) = n \log(n+1) - 1.193n + O(\log n)$  comparisons and  $\Theta(n \log n)$  operations on the average.*

### 3 Chen's Algorithm

We slightly generalise Chen's algorithm to a three parameter family of algorithms, with parameters  $\gamma$ ,  $\lambda$ , and  $\mu$ , described below. Typically,  $\gamma \geq 2$  and a power of 2,  $\lambda \geq 2$  and a power of 2,  $\lambda \leq \mu$ , and  $\gamma \leq \mu$ ; Chen [Che02] had considered a one parameter family with  $\gamma = \lambda$  and  $\mu = \lambda(\lambda - 1)$ , for arbitrary  $\lambda \geq 2$ .

As with the partition sorts, the sort begins with a recursive sort of the first  $\left\lfloor \frac{n}{\gamma} \right\rfloor$  items. The heart of the algorithm is provided by *Chen's sort completion procedure*, for completing the sort of a partially sorted array  $(S, U)$ ; as before,  $S$  is sorted and  $U$  is unsorted. This recursive procedure is similar to that used in the partition sort, except for proceeding differently when  $S$  is small relative to  $U$ , and has the following form:

Case  $\mu(s+1) - 1 \geq (s+u)$ : Thus,  $(\mu-1)(s+1) \geq u$ ; we say that such an input has the *balanced property*.

First, the unsorted subarray  $U$  is partitioned about the median  $x$  of  $S$ , resulting in blocks  $U_L$  and  $U_R$ ; next, the blocks  $\{x\} \cup S_R$  and  $U_L$  are swapped, preserving the order of items in  $\{x\} \cup S_R$ , but not necessarily of those in  $U_L$ , yielding the ordering  $S_L U'_L x S_R U_R$  with  $S_L \cup U'_L < x < S_R \cup U_R$ ; finally the subarrays  $(S_L, U'_L)$  and  $(S_R, U_R)$  are sorted recursively.

Case  $\mu(s+1) - 1 < (s+u)$ : Thus,  $(\mu-1)(s+1) < u$ ; we call this case *expansion*, and in particular for  $\lambda = 2$ , *doubling*.

First, the sorted subarray  $S$  is expanded as follows: Let  $S'$  be the leftmost subarray of size  $\lceil \lambda(s+1) - 1 \rceil$ . The subarray  $S'$  is sorted recursively, with  $S$  as the sorted subarray; then, the sort of the whole array continues recursively, with  $S'$  as the sorted subarray.

Chen's description [Che02] of his algorithm can be viewed as bottom-up;  $(s, u)$  is set to  $(1, n-1)$  initially, which is analogous to setting  $\gamma = \lambda$ ; we have replaced his single parameter with three.

#### 3.1 Average Case Analysis of Comparisons

We prove that on average the partition sort with parameter  $\gamma$  performs at least as many comparisons as (*dominates*) Chen's algorithm with parameters  $(\lambda, \mu, \gamma)$ . We analyse the case  $\lambda = 2$  in detail.

For purposes of the analysis, we introduce another sorting algorithm, which we call the *variant partition sort* (to contrast it with the original partition sort, henceforth qualified as *basic* to emphasise the distinction). We prove two claims:

**Claim 1.** *The basic partition sort dominates the variant partition sort.*

**Claim 2.** *If Claim 1 holds, then the basic partition sort dominates Chen's algorithm.*

It is helpful to view the basic partition sort as follows, when its input is an unbalanced problem instance  $(S, U)$  with  $(\mu - 1)(s + 1) < u$  (where  $s = |S|$  and  $u = |U|$ ); we name the  $s$  items in  $S$  type  $A$  items, the immediately following  $s + 1$  items in  $U$  type  $B$  items, and the remaining  $t \stackrel{\text{def}}{=} u - (s + 1)$  items in  $U$  type  $C$  items; the basic partition sort then proceeds as follows:

- P1: It places each of the  $u$  items of  $U$  in one of the  $s + 1$  buckets delimited by items in  $S$ ; we call these the  $A$  buckets. Each placement takes  $\log(s + 1)$  comparisons.
- P2: It sorts each of the  $s + 1$   $A$  buckets by insertion sort.

The variant partition sort, which mimics Chen's algorithm to a certain extent, proceeds as follows:

- V1: It places each of the  $s + 1$  type  $B$  items of  $U$  in one of the  $s + 1$   $A$  buckets delimited by items in  $S$ . Each placement takes  $\log(s + 1)$  comparisons.
- V2: It sorts each of the  $s + 1$   $A$  buckets by insertion sort.
- V3: It places each of the  $t$  type  $C$  items of  $U$  in one of the  $2s + 2$  buckets delimited by items in  $A \cup B$ ; we call these the  $AB$  buckets. Each placement takes  $1 + \log(s + 1)$  comparisons.
- V4: It sorts each of the  $2s + 2$   $AB$  buckets by insertion sort.

**Lemma 5.** *For  $\lambda = 2$ , Claim 2 holds.*

*Proof.* We proceed by induction on  $s + u$ . Clearly, for  $s + u = 1$ , the result holds.

For  $s + u > 1$ , if  $(\mu - 1)(s + 1) \geq u$ , the initial call in Chen's algorithm is a non-doubling call, and hence the same as in the partition sort; for this case the result follows by induction applied to the recursive calls.

It remains to consider the unbalanced case with  $(\mu - 1)(s + 1) < u$ . Here the initial call in Chen's algorithm is a doubling call. In the variant partition sort, the same doubling call occurs, but thereafter it performs no more doubling calls, i.e., in its recursive calls on subproblems  $(\tilde{s}, \tilde{u})$  it acts in the same way as the basic partition sort on input  $(\tilde{s}, \tilde{u})$ . We consider the two top level recursive calls generated by the doubling call  $(s, s + 1)$ , namely  $(\frac{s-1}{2}, u_1)$  and  $(\frac{s-1}{2}, s + 1 - u_1)$ , and the call after the doubling completes,  $(2s + 1, u - s - 1)$ . For each value of  $u_1$ , all possible orderings of the items are equally likely, given that the initial  $(s, u)$  problem was distributed uniformly at random. Consequently, the inductive hypothesis can be applied to these three inner calls, showing that the basic partition sort dominates Chen's algorithm in each of these inner calls. Consequently, the variant partition sort dominates Chen's algorithm for the original  $(s, u)$  call. Given our assumption that the basic partition sort dominates the variant partition sort, this proves the inductive step.  $\square$

**Lemma 6.** *For  $\lambda = 2$ , Claim 1 holds.*

*Proof.* To facilitate the comparison of the basic partition sort with its variant, we view the execution of the basic partition sort in the following alternative way; this form has exactly the same comparison cost as the previous description.



- P1': It places each of the  $s + 1$  type  $B$  items of  $U$  in one of the  $s + 1$   $A$  buckets delimited by items in  $S$ . (Each placement takes  $\log(s + 1)$  comparisons.) This is identical to step V1.
- P2': It sorts each of the  $s + 1$   $A$  buckets (containing type  $B$  items) by insertion sort. This is identical to step V2.
- P3': It places each of the  $t$  type  $C$  items of  $U$  at the right end of one of the  $s + 1$   $A$  buckets in  $A \cup B$  (which already contain type  $B$  items in sorted order). (Each placement is done by a binary search over the  $A$  items, taking  $\log(s + 1)$  comparisons.)
- P4': For each of the  $s + 1$   $A$  buckets, it places the type  $C$  items in their correct position by insertion sort (at the start of this step, each such  $A$  bucket contains both type  $B$  and type  $C$  items, with type  $B$  items in sorted order, and type  $C$  items at the right end).

It remains to determine the relative costs of steps V3 and V4 and of steps P3' and P4'. Step V3 performs one more comparison for each type  $C$  item, for a total excess of  $t$  comparisons (this is clear for  $s = 2^k - 1$ , and needs a straightforward calculation otherwise). In step V4, the only comparisons are between type  $C$  items within the same  $AB$  bucket, whereas in step P4' there are, in addition, comparisons between type  $C$  items in distinct  $AB$  buckets (but in the same  $A$  bucket), and also comparisons between type  $B$  and type  $C$  items in the same  $A$  bucket, called  $BC$  comparisons. We confine our attention to  $BC$  comparisons and show that there are at least  $t$  such comparisons, on the average; it would then follow that on average the partition sort dominates its variant.

If it were the case that one type  $B$  item ( $s + 1$  in number) went into each of the  $s + 1$   $A$  buckets, resulting in an *equipartition*, it is evident that the number of  $BC$  comparisons would be exactly  $t$ . We now argue that this is in fact a minimum, when considering comparisons on the average.

Given the  $AB$  sequence with equipartition (i.e., with one  $B$  item in each  $A$  bucket) consider perturbing it to a different sequence. For any such non-trivial perturbation, there is at least one  $A$  bucket which holds a group of  $r + 1 > 1$  type  $B$  items (and which contains  $r + 2$   $AB$  buckets).

Consider an  $A$  bucket with multiple type  $B$  items in the perturbed sequence. The  $r$  excess type  $B$  items must have been drawn from (originally) singleton  $A$  buckets (which originally contained two  $AB$  buckets) and thus there are  $r$  empty  $A$  buckets (which now contain only one  $AB$  bucket). An arbitrary type  $C$  item could go into any of the  $2s + 2$   $AB$  buckets with equal probability. The number of  $BC$  comparisons that this type  $C$  item undergoes, on the average, increases by:

$$\frac{1}{2s + 2} \left[ -2 \cdot (r + 1) + (r + 2) \cdot \left( \frac{r + 3}{2} - \frac{1}{r + 2} \right) \right] > 0$$

for  $r \geq 1$  as a consequence of the perturbation.

Finally, we note that in the perturbed sequence, for each  $A$  bucket with  $r + 1 > 1$  type  $B$  items, the number of  $BC$  comparisons an arbitrary type  $C$  item undergoes increases, on the average, by the above quantity.

Thus, the equipartition configuration minimises the number of  $BC$  comparisons, and consequently, the basic partition sort dominates its variant.  $\square$

Setting  $\gamma = 2$  to exhibit a bound, from dominance and Theorem 2 we have:

**Theorem 3.** *If all input orderings are equally likely, Chen’s algorithm, with  $(\lambda, \gamma) = (2, 2)$ , performs at most  $n \log n - (\frac{1}{2} + \ln 2) n + O(\log n)$  comparisons on the average, independently of  $\mu$ .*

### 3.2 Operation Counts

**Theorem 4.** *Chen’s algorithm performs  $\Theta(\frac{\mu}{\lambda} \log \lambda \, n \log n)$  comparisons in the worst case, independently of  $\gamma$ .*

**Theorem 5.** *Chen’s algorithm performs  $\Theta(n \frac{\log^2 n}{\log^2 \frac{\mu}{\lambda}})$  exchanges in the worst case.*

**Theorem 6.** *If all input orderings are equally likely, Chen’s algorithm with  $\lambda(1 + \epsilon) \leq \mu$  performs  $\Theta(\gamma \, n \log n)$  exchanges on the average.*

## 4 Empirical Studies

We implemented and compared the performance of quicksort, the partition sort and Chen’s algorithm. We measured the running times  $T_n$  and counted comparisons  $C_n$  and data moves  $M_n$ , for various input sizes  $n$ .

We implemented quicksort in two different ways. The first largely follows Sedgewick [Sed78]; it uses a best-of-three strategy for selecting the pivot [Hoa62, Sin69], and sorts small subproblems (of size less than the insertion cutover) using insertion sort, but unlike Sedgewick, performs the insertion sorts as they arise (locally), rather than in one final pass (globally). Our experiments showed that for large input sizes, the local implementation yielded a speedup. The second implementation is similar except that it uses Tukey’s ‘ninther’, the median of the medians of three samples (each of three items), as the pivot; this quasi-best-of-nine version due to Bentley and McIlroy [BM93] was observed to be about 3% faster than the best-of-three version.

For the partition sort and Chen’s algorithm, the block swaps (of blocks  $\{x\} \cup S_R$  and  $U_L$ ) are performed using Chen’s optimised implementation [Che96].

Best results for the partition sort were obtained with  $\gamma$  about 128. The average size of the bucket is then quite large and we found it to be significantly more efficient to sort them with quicksort. Moderate variations of  $\gamma$  had little impact on performance.

Again, with Chen’s algorithm, we added a cutover to quicksort on moderate sized subproblems.<sup>4</sup> Our best performance arose with  $\lambda = 2$ ,  $\mu = 128$ , and

<sup>4</sup> Without this cutover we were unable to duplicate Chen’s experimental results [Che02]. He reported a 5-10% speedup compared to best-of-three quicksort on inputs of size up to 50,000.

$\gamma = 128$ , and a quicksort cutover of about 500. Again, moderate variation of  $\mu$  and  $\gamma$  had little effect on the performance.

We compared top-down and bottom-up drivers in our experiments, and found that top-down drivers tend to have smoother behaviour. We have therefore chosen to use top-down drivers in our implementations.

For our algorithms, we resort to standard hand optimisations, such as eliminating the tail recursions by branching (and other recursions by explicit stack management); we also inline the code for internal procedures.

We ran each algorithm on a variety of problem sizes. Each algorithm, for each problem size, was run on the same collection of 25 randomly generated permutations of integers (drawn from the uniform distribution). Running times for the best choices of parameters are shown in the table below.

RUNNING TIMES $\overline{T_n} \pm \sigma(T_n)$ ( $\mu$ s)				
$n$	Partition	Chen	Quasi-best-of-9	Best-of-3
20000	4906 $\pm$ 110	4878 $\pm$ 78	4840 $\pm$ 190	4868 $\pm$ 182
25000	6306 $\pm$ 107	6421 $\pm$ 185	6134 $\pm$ 183	6268 $\pm$ 183
30000	7652 $\pm$ 78	7863 $\pm$ 150	7493 $\pm$ 241	7805 $\pm$ 201
35000	9124 $\pm$ 200	9155 $\pm$ 217	8969 $\pm$ 315	9215 $\pm$ 252
40000	10577 $\pm$ 149	10689 $\pm$ 221	10406 $\pm$ 235	10601 $\pm$ 284
200000	65453 $\pm$ 764	67048 $\pm$ 7861	65294 $\pm$ 1467	87997 $\pm$ 6412
250000	84526 $\pm$ 1355	84551 $\pm$ 1269	84110 $\pm$ 914	86294 $\pm$ 2199
300000	103683 $\pm$ 354	104319 $\pm$ 3631	102909 $\pm$ 1264	105514 $\pm$ 2124
350000	123258 $\pm$ 1457	124211 $\pm$ 2542	122788 $\pm$ 1389	127300 $\pm$ 2900
400000	143115 $\pm$ 1024	144246 $\pm$ 2007	143029 $\pm$ 1622	147475 $\pm$ 2328
2000000	858194 $\pm$ 3038	862604 $\pm$ 5364	893321 $\pm$ 2779	897493 $\pm$ 6893
2500000	1093029 $\pm$ 2805	1100885 $\pm$ 4665	1107729 $\pm$ 12688	1148407 $\pm$ 10356
3000000	1344032 $\pm$ 9178	1348160 $\pm$ 9513	1360892 $\pm$ 2680	1404965 $\pm$ 11242
3500000	1583731 $\pm$ 5295	1596032 $\pm$ 8990	1602471 $\pm$ 9326	1663999 $\pm$ 9192
4000000	1833468 $\pm$ 4975	1845352 $\pm$ 9949	1854723 $\pm$ 12678	1921886 $\pm$ 2685

In our experiments, even for trials repeated only a few times, the deviations are orders of magnitude smaller than the average. This is most striking for the operation counts (not shown) where, with trials repeated 25 times, the deviations are typically below 0.1% of the averages for the new algorithms, and below 2% for the quicksorts.

The experiments were conducted on a Sun UltraSPARC-IIe processor with CPU speed 550 MHz and Cache size 512 KB. The programs, written in C, were run under Sun OS (kernel version Generic\_112233-08) with gcc (version 3.3.3) compilation at -O3 level. Broadly similar results were obtained on a Pentium running GNU/Linux.

**Acknowledgements.** We thank J. Ian Munro, Alan Siegel, and an anonymous referee for their helpful comments.

## References

- [Ben00] Jon L. Bentley. *Programming Pearls*. Addison-Wesley, Reading, MA, second edition, 2000.
- [BM93] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Software Practice and Experience*, 23(11):1249–1265, November 1993.
- [Che96] Jing-Chao Chen. Proportion split sort. *Nordic Journal of Computing*, 3(3):271–279, Fall 1996.
- [Che02] Jing-Chao Chen. Proportion extend sort. *SIAM Journal on Computing*, 31(1):323–330, February 2002.
- [Dut93] Ronald D. Dutton. Weak-heapsort. *BIT*, 33(3):372–381, 1993.
- [EW00] Stefan Edelkamp and Ingo Wegener. On the performance of weak-heapsort. *Lecture Notes in Computer Science*, 1770:254–266, 2000.
- [FG03] Gianni Franceschini and Viliam Geffert. An in-place sorting with  $O(n \log n)$  comparisons and  $O(n)$  moves. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 242–250, Cambridge, Massachusetts, 11–14 October 2003.
- [FJ59] Lester R. Ford, Jr., and Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66(5):387–389, May 1959.
- [Flo64] Robert W. Floyd. ACM Algorithm 245: Treesort 3. *Communications of the ACM*, 7(12):701, December 1964.
- [Hoa61] C. A. R. Hoare. ACM Algorithm 63: Partition, ACM Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321–322, July 1961.
- [Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, April 1962.
- [Knu73] Donald E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.
- [KPT96] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. Practical in-place mergesort. *Nordic Journal of Computing*, 3(1):27–40, Spring 1996.
- [LL97] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, New Orleans, Louisiana, 5–7 January 1997.
- [Mus97] David R. Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27(8):983–993, August 1997.
- [Rei85] Rüdiger Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing*, 14(2):396–409, May 1985.
- [Rei92] Klaus Reinhardt. Sorting in-place with a worst case complexity of  $n \log n - 1.3n + o(\log n)$  comparisons and  $\epsilon n \log n + o(1)$  transports. In *Algorithms and Computation, Third International Symposium, ISAAC '92, Proceedings*, pages 489–498, Nagoya, Japan, 16–18 December 1992.
- [Sed78] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, October 1978.
- [Sin69] R. C. Singleton. An efficient algorithm for sorting with minimal storage. *Communications of the ACM*, 12(3):185–187, March 1969.
- [Weg93] Ingo Wegener. Bottom-up heapsort, a new variant of heapsort, beating, on an average, quicksort (if  $n$  is not very small). *Theoretical Computer Science*, 118(1):81–98, 13 September 1993.
- [Wil64] J. W. J. Williams. ACM Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, June 1964.

# A Fast Distributed Algorithm for Approximating the Maximum Matching<sup>\*</sup>

Andrzej Czygrinow<sup>1</sup>, Michał Hańćkowiak<sup>2</sup>, and Edyta Szymańska<sup>2</sup>

<sup>1</sup> Department of Mathematics and Statistics  
Arizona State University  
Tempe, AZ 85287-1804, USA  
`andrzej@math.la.asu.edu`

<sup>2</sup> Faculty of Mathematics and Computer Science  
Adam Mickiewicz University  
Poznań, Poland  
`{mhanckow,edka}@amu.edu.pl`

**Abstract.** We present a distributed approximation algorithm that computes in every graph  $G$  a matching  $M$  of size at least  $\frac{2}{3}\beta(G)$ , where  $\beta(G)$  is the size of a maximum matching in  $G$ . The algorithm runs in  $O(\log^4 |V(G)|)$  rounds in the synchronous, message passing model of computation and matches the best known asymptotic complexity for computing a maximal matching in the same protocol. This improves the running time of an algorithm proposed recently by the authors in [2].

## 1 Introduction

Any set of pairwise disjoint edges constitutes a matching of a given graph. From the algorithmic point of view two basic problems arise in the context of matchings. One is to compute any matching of maximum cardinality (maximum matching), the second one is to construct a matching not contained in any other matching of a given graph (inclusion maximal matching). There is rich literature concerning sequential algorithms for the maximum matching problem (see Lovász and Plummer [5] for an excellent survey). Several sequential algorithms for general graphs which are polynomial in time are known and many of them use augmenting paths as a main tool. It has turned out, however, that even the very simple sequential greedy algorithm is not known to be implemented in parallel. Many problems become even more difficult when the parallel network is not equipped with the shared memory storage. The increasing importance of large-scale networks such as Internet, ad-hoc and sensor networks have motivated active research on distributed computing in a message passing systems. Adopting the existing sequential procedures results in a rather very inefficient scheme and a new algorithmic approach is needed.

We consider the distributed model of computation introduced by Linial in [6]. In this model a network is represented by an undirected graph where each

---

<sup>\*</sup> Research supported by KBN grant no. 7 T11C 032 20

vertex stands for a processor and each edge corresponds to a connection between two processors in the network. Each processor has a unique ID and knows the number of vertices in the graph but not the global topology of the network. We assume full synchronization of the network: computation is performed in steps. In a single step, each processor can send a message to all of its neighbors, collect messages from its neighbors, and perform some local computations. Note that the above model is completely different from the parallel PRAM model where each computational unit can communicate with all other units at each step of the computations. As a result many problems that admit efficient algorithms in the PRAM model are still open in the distributed model. An eminent example, which is a generalization of the problem studied in this work, is the Maximal Independent Set (MIS) Problem. Efficient  $\mathcal{NC}$  algorithms for MIS are well-known ([7]) but the question if there is an efficient distributed algorithm for the problem is one of the main open problems in the area ([6]). However, an efficient distributed algorithm for the related Maximal Matching (MM) Problem is known due to Hańćkowiak, Karoński, and Panconesi. In [4] the three authors proved the following theorem.

**Theorem 1 ([4]).** *There is a distributed procedure MATCH which in  $O(\log^4 n)$  many communication rounds in the synchronous, distributed model of computation, computes a maximal matching  $M$  in any graph  $G$  on  $n$  vertices.*

The procedure MATCH is a starting point for further research in [2] and in this paper. In [2], building on ideas from [4], the authors designed an algorithm which finds a maximal matching of size which is relatively close to the maximum:

**Theorem 2 ([2]).** *There is a distributed algorithm which in  $O(\log^6 n)$  steps finds a matching  $M$  in any graph  $G$  on  $n$  vertices, such that*

$$|M| \geq \frac{2}{3}\beta(G)$$

where  $\beta(G)$  denotes the size of the largest matching in  $G$ .

In this paper, we propose a substantial modification of the algorithm from Theorem 2 which runs in  $O(\log^4 n)$  steps and therefore has the same time complexity as the algorithm MATCH in Theorem 1. The following theorem is proved:

**Theorem 3.** *There is a distributed procedure MATCHING which in  $O(\log^4 n)$  steps finds in any graph  $G(V, E)$  with  $n = |V(G)|$  vertices a matching  $M$ , such that*

$$|M| \geq \frac{2}{3}\beta(G).$$

Although the approximation ratio of our result may not be very appealing, it should be remembered that the problem of computing an optimal solution in a general graph is even not known to be in  $\mathcal{NC}$ . And yet the approximation algorithm given by [3] is inherently designed for the EREW PRAM.

Interestingly, the result is purely deterministic and together with [4] stands among few examples of distributed computing, where polylogarithmic time complexity is achieved without the use of random bits. The  $O(\log^6 n)$  complexity of the algorithm from Theorem 2 comes from sequential iterations over  $O(\log^2 n)$  blocks (defined in the next section). It is this part of the algorithm which we improve. The modification that we present allows us to perform computations in parallel and as a result we drop the  $O(\log^2 n)$  factor in the time complexity altogether.

Following the strategy from [2], we compute a maximal matching  $M$  and then a maximal set of vertex-disjoint paths of length three that augment  $M$ . After augmenting the matching  $M$  along these paths we obtain a new matching  $M'$  such that  $|M'| \geq \frac{2}{3}\beta(G)$ . To find a maximal matching we invoke the procedure MATCH from Theorem 1 and to compute a maximal set of disjoint paths we design a new procedure, which is the main contribution of this paper.

In the next section, we present definitions and notation which are necessary to develop the algorithm together with some preliminary lemmas. The last section contains the description of the algorithm and a sketch of the proof of its correctness. For complete proofs the reader is referred to the full version of this paper.

## 2 Definitions, Notation, and Preliminary Lemmas

In this section we fix some notation, introduce terminology, and state a few useful lemmas. As our algorithm creates an auxiliary multigraph, we define most of the terms in the multigraph setting.

Let  $M$  be a matching in a (multi)graph  $G = (V, E)$ . We say that a vertex  $v$  is *M-saturated* if  $v$  is an endpoint of some edge from  $M$ . An edge  $e = \{u, v\}$  is *M-saturated* if either  $u$  or  $v$  is *M-saturated*. A path  $P$  is *M-alternating* if it contains alternately edges from  $M$  and from  $E \setminus M$ . A path  $P$  of length  $2k + 1$ ,  $k \geq 0$ , *augments*  $M$  if  $P$  is *M-alternating* and no end of  $P$  is *M-saturated*. A special role in the paper will be played by paths of length three augmenting  $M$ . A path is called an *(M, 3)-path* if it augments  $M$  and has length three. A set of paths is called *independent* if every two paths from the set are vertex-disjoint.

Just like in [2], our algorithm is based on the following facts that are standard in graph theory.

### Claim 4

- (a) *Let  $M$  be an arbitrary matching in  $G$ . If there are no paths of length at most three augmenting  $M$ , then*

$$|M| \geq \frac{2}{3}\beta(G).$$

- (b) *Let  $M$  be a maximal matching in a graph  $G$  and let  $\mathcal{P}$  be a maximal independent set of paths of length three augmenting  $M$ . Further, let  $M' = M \div (\bigcup_{P \in \mathcal{P}} E(P))$  be the matching obtained by augmenting  $M$  along the paths from  $\mathcal{P}$ . Then there are no paths of length at most three augmenting  $M'$  in  $G$ .*

Consequently, parts (a) and (b) of Claim 4 together assure, that the matching  $M'$  satisfies

$$|M'| \geq \frac{2}{3}\beta(G).$$

In the process of constructing the matching we will use the notion of a *substantial* matching.

**Definition 1.** Let  $M$  be a matching in a multigraph  $G = (V, E)$  and  $\gamma > 0$ . A matching  $M$  is  $\gamma$ -substantial in  $G$  if the number of  $M$ -saturated edges of  $G$  is at least  $\gamma|E|$ .

Using an algorithm from [4] (with minor changes), we can find a substantial matching in a bipartite multigraph.

**Lemma 1.** For all constants  $C > 0$  and  $0 < \gamma < 1$  there exists a distributed algorithm that finds in  $O(\log^3 n)$  steps a  $\gamma$ -substantial matching in a bipartite multigraph  $G = (L, R, F)$ , where  $|F| \leq n^C$  and  $|L| + |R| = n$ .

Similarly, we will also use the notion of a substantial set of paths. Let  $\mathcal{P}_3(M)$  denote the set of all  $(M, 3)$ -paths in a graph  $G = (V, E)$ .

**Definition 2.** Let  $M$  be a matching in  $G$  and  $\gamma > 0$ . A set  $\mathcal{P}$  of  $(M, 3)$ -paths is called  $\gamma$ -path-substantial in  $G$  if the number of paths from  $\mathcal{P}_3(M)$  that have a common vertex with some path in  $\mathcal{P}$  is at least  $\gamma|\mathcal{P}_3(M)|$ .

Now we will introduce more technical terminology. Given a bipartite multigraph  $H = (L, R, E)$  we partition its left-hand side  $L$  into sets  $L_i = \{u \in L : \frac{D_i}{2} < \deg_H(u) \leq D_i\}$  for  $D_i = 2^i$  and  $i \geq 0$ . The submultigraph of  $H$  induced by the edges incident to  $L_i$ , denoted by  $H_i = (L_i, N(L_i), E_i)$  is called a  $D_i$ -block. A key concept which will be used in our approach is the concept of a spanner.

**Definition 3.** Let  $H = (A, B, E)$  be a  $D$ -block for some  $D$  as defined above. An  $(\alpha, K)$ -spanner from  $A$  to  $B$  is a subgraph  $S = (A', B, E')$  of  $H$  such that the following conditions are satisfied.

1.  $|A'| \geq \alpha|A|$ .
2. For every vertex  $a \in A'$ ,  $\deg_S(a) = 1$ .
3. For every vertex  $b \in B$ ,  $\deg_S(b) < \frac{K}{D}\deg_H(b) + 1$ .

In other words, a spanner is a collection of stars such that the degrees of the centers of the stars are appropriately bounded in terms of their degrees in  $H$ .

Note that spanners played an important role in designing the algorithm for maximal matching in [4]. Although the procedures in [4] are formulated for simple graphs they can be adopted to multigraphs with only minor changes. In particular, we have the following fact.

**Lemma 2 ([4]).** For every constant  $C$  and input parameter  $D$  there exist a constant  $K = K(C)$  and a distributed algorithm that finds in  $O(\log^3 n)$  steps a  $(\frac{1}{2}, K)$ -spanner in a multigraph  $H = (A, B, E)$  that is a  $D$ -block with  $|E| \leq n^C$  and  $n = |A| + |B|$ .



Specifically, in our case  $C = 4$  and  $K < 100$ .

In order to motivate the following definitions, we spend some time on explaining the main idea of our algorithm.

The approach used to find a set of augmenting paths is based on the following strategy. First we find a maximal matching  $M$  using the procedure MATCH given in Theorem 1. Then the maximal independent set of  $(M, 3)$ -paths is gradually constructed in rounds. In each round a substantial set of  $(M, 3)$ -paths is found. It turns out that we can find a substantial set of  $(M, 3)$ -paths in a special layered graph. For that reason, from the input graph, we obtain a virtual auxiliary graph which is of the form. Finally, once the paths are found in the layered graph, we translate them back to the original graph.

The layered graph will be a simple graph which has four layers of vertices and we will refer to it as a *4L-graph*. A precise construction of a 4L-graph is given in procedure REDUCE placed in the next section and here we just introduce its structural features. Every 4L-graph  $H = H(G, M)$ , with respect to  $G$  and a maximal matching  $M$ , has the vertex set  $V(H)$  partitioned into four nonempty sets  $X_1, X_2, X_3, X_4$  so that  $H[X_2, X_3] = M$  and every vertex from  $X_1$  is adjacent only to vertices from  $X_2$  and every vertex from  $X_4$  is adjacent only to vertices from  $X_3$ . In other words,  $H$  consists of 3 bipartite graphs. The edges of  $H[X_1, X_2]$  and  $H[X_3, X_4]$  are potential ingredients of the  $(M, 3)$ -paths extending  $M$ . The desired structure of a 4L-graph is obtained by first removing from  $E(G)$  all edges which violate the above properties. We will distinguish in the input graph  $G$  three types of edges with respect to  $M$ :

- (a) edges that are in  $M$ ,
- (b) edges that have exactly one endpoint in  $M$  (these edges may form triangles based on an edge from  $M$ ),
- (c) edges that are not in  $M$  but have both endpoints in  $M$ .

Note that edges from (c) do not belong to any  $(M, 3)$ -path of  $G$  and therefore we can delete them. Next, the triangles built of edges of type (b) are carefully destroyed (see procedure REDUCE). Finally, some vertices from  $V(G) \setminus V(M)$  are replicated to have their counterparts in  $X_4$  and  $X_1$ , respectively.

Let  $H = (X_1, X_2, X_3, X_4, E)$  be a 4L-graph. Now we introduce an auxiliary multigraph  $Mul(H) = (X_1, X_2, X_3, X_4, F)$  defined as follows.

**Definition 4.** *Let  $H = (X_1, X_2, X_3, X_4, E)$  be a 4L-graph and for every  $e \in H[X_1, X_2] \cup H[X_3, X_4]$ , let  $w_e$  denote the number of  $(M, 3)$ -paths in  $H$  containing  $e$ . The multigraph  $Mul(H) = (X_1, X_2, X_3, X_4, F)$  is a graph obtained from  $H$  by putting  $w_e$  copies of  $e$  in  $F$  for every edge  $e \in H[X_1, X_2] \cup H[X_3, X_4]$ , and one copy of  $e$  for every edge  $e \in H[X_2, X_3]$  in  $F$ .*

The main property of  $Mul(H)$  is that edges in  $Mul(H)[X_1, X_2]$  and similarly, in  $Mul(H)[X_3, X_4]$  correspond to  $(M, 3)$ -paths in  $H$ . In the course of our proof we will make use of it by finding a substantial matching in  $Mul(H)[X_1, X_2]$  and deducing a substantial property for a set of  $(M, 3)$ -paths built on the matching. Directly from Definition 4 we can derive the following fact.

**Fact 5.** For every  $e = \{a, a'\} \in H[X_2, X_3]$ ,  $\deg_{Mul(H)}(a) = \deg_{Mul(H)}(a')$  which is equal to the number of  $(M, 3)$ -paths containing  $e$ . Moreover, there is a one-to-one correspondence between the edges in  $Mul(H)[X_1, X_2]$  and the  $(M, 3)$ -paths in  $H$ .

To this end, in  $Mul(H)$ , we define a  $4L$ - $D$ -block. The actual process of constructing the set of  $(M, 3)$ -paths will be performed in blocks that will partition the sets  $X_2$  and  $X_3$ .

**Definition 5.** Let  $R$  be the set of all edges  $e = \{a, a'\} \in Mul(H)[X_2, X_3]$  that satisfy

$$\frac{D}{2} < \deg_{Mul(H)}(a) = \deg_{Mul(H)}(a') \leq D$$

and let  $X'_2 = (\bigcup_{e \in R} e) \cap X_2$ ,  $X'_3 = (\bigcup_{e \in R} e) \cap X_3$ . A  $4L$ -sub-multigraph  $B_D(H) = (X_1, X'_2, X'_3, X_4, F')$  of  $Mul(H) = (X_1, X_2, X_3, X_4, F)$  which contains all edges incident to edges from  $R$  is called a  $4L$ - $D$ -block.

It is worth noticing here, that considering  $D = D(i) = 2^i$  for  $i = 0, 1, \dots, \log n$ ,  $4L$ -blocks  $B_{D(i)}$  are edge disjoint and partition the sets  $X_2$  and  $X_3$ . What is more, every subgraph  $B_{D(i)}(H)[X_3, X_4]$  is a  $D(i)$ -block.

### 3 Algorithm

In this section we present the main algorithm. In each iteration it constructs a substantial set of  $(M, 3)$ -paths and adds them to the global set of independent  $(M, 3)$ -paths. Then the paths are removed from the graph together with all edges incident to them. It stops when no such paths are left in the graph and outputs a new matching  $M^*$ . Below we present an outline of the algorithm to give some intuition behind our approach.

---

#### Procedure Matching

---

Input: Graph  $G$

Output: Matching  $M^*$

1. Find a maximal matching  $M$  in  $G$ .
  2. For  $j := 1$  to  $O(\log n)$  do
    - a) Construct a  $4L$ -graph  $\bar{G} = \bar{G}(G, M) = (X_1, X_2, X_3, X_4, E)$  from  $G$  and  $M$ .
    - b) Construct a multigraph  $\tilde{G} = Mul(\bar{G}) = (X_1, X_2, X_3, X_4, F)$  with multiple edges in  $\bar{G}[X_1, X_2]$  and  $\bar{G}[X_3, X_4]$ .
    - c) Find a  $\xi$ -path-substantial set  $\tilde{\mathcal{P}}$  of disjoint  $(M, 3)$ -paths in  $\tilde{G}$  using spanners in blocks in parallel.
    - d) Translate  $\tilde{\mathcal{P}}$  found in  $\tilde{G}$  to  $\mathcal{P}$  in  $G$ .
    - e) Accumulate the  $(M, 3)$ -paths from  $\mathcal{P}$ .
    - f) Update  $G$  (remove from  $G$  the paths in  $\mathcal{P}$  with all edges incident to them).
  3. Update  $M^*$  by augmenting  $M$  along the paths from  $\mathcal{P}$ .
-

The algorithm consists of a procedure that computes a substantial matching, and two procedures from [2]. First we describe a new procedure which computes a substantial set of  $(M, 3)$ -paths in a given 4L-graph. Then we recall two procedures from [2]; the first one reduces a graph  $G = (V, E)$  and a maximal matching  $M$  in  $G$  to a 4L-graph  $\bar{G} = (X_1, X_2, X_3, X_4, E)$ , the second one translates a  $\tilde{\alpha}$ -substantial set  $\tilde{\mathcal{P}}$  of independent  $(M, 3)$ -paths in  $\bar{G}$  to  $\alpha$ -substantial set  $\mathcal{P}$  of independent  $(M, 3)$ -paths in  $G$ .

### 3.1 Algorithm in a 4L-Graph

In this section, we present the main part of our algorithm, PROCEDURE PATHS-IN4LGRAPH. This procedure finds a  $\xi$ -path-substantial set  $\mathcal{P}$  of disjoint  $(M, 3)$ -paths in a layered graph. We are building the paths in two phases. In the first phase we choose candidates for edges in  $\bar{G}[X_3, X_4]$  which will augment the matching  $M$ . This is performed by first constructing a star forest with all stars having centers in  $X_4$ . After the second phase at most one edge from every star is added to the set of  $(M, 3)$ -paths. Therefore we move along the star rays to the set  $X_2$  and introduce an auxiliary, bipartite graph with vertices corresponding to the stars we just created, on one side and  $X_1$  on the other. In the second phase we move our attention to this graph and find a substantial matching in it. After extending uniquely the matching to the graph  $\bar{G}[X_3, X_4]$  we obtain the set  $\mathcal{P}$  of disjoint  $(M, 3)$ -paths.

Phase A: Steps 1-5 construct a star forest  $S'$  in  $H[X_3, X_4]$ .

Phase B: Steps 6-8 construct the set  $\mathcal{P}$  using  $S'$ .

Note that Phase B is analogous to the main algorithm for the 4L-graph in [2]. However, since the blocks in PATHSIN4LGRAPH are defined in a new way compared with [2], the analysis is different. The main problem in PATHSIN4LGRAPH is to design the star forest  $S'$  so that the stars can be glued together and used as super-vertices in Phase B. Properties of  $S'$  which we need to prove that the algorithm is indeed correct are quite delicate, on one hand we must delete some edges of a union of spanners  $S$  to make the analysis work, on the other we need to keep most of them to retain the substantial property. In particular, if a star has exactly one vertex in a block then it must have no vertices in any other block. This is taken care of in steps (4) and (5) of the procedure given below.

---

#### Procedure PathsIn4LGraph

---

Input: 4L-graph  $\bar{G} = (X_1, X_2, X_3, X_4, E)$ , where  $\bar{G}[X_2, X_3] = M$ .

Output: A set  $\mathcal{P}$  of disjoint  $(M, 3)$ -paths which is  $\xi$ -path-substantial in  $\bar{G}$  for some constant  $\xi > 0$ .

1. Construct the 4L-multigraph  $\tilde{G} := \text{Mul}(\bar{G})$  as in Definition 4.
2. For  $i = 0$  to  $4 \log n$  do  $D(i) := 2^i$

$B_i := B_{D(i)}(\tilde{G}) = (X_1, X_2(i), X_3(i), X_4, E_i)$ ,

where  $X_j(i)$  is a subset of  $X_j$  corresponding to the block  $B_i$ , for  $j = 1, 2$ .

3. In parallel, for every  $i$ : Let  $K = K(4)$  be a constant in Lemma 2. Find a  $(\frac{1}{2}, K)$ -spanner  $S_i$  from  $X_3(i)$  to  $X_4$  in  $B_i[X_3(i), X_4]$  (using the procedure from [4]).
4. Let  $S = \bigcup_{i=0}^{4 \log n} S_i$  be a star forest.
  - a) For any star  $Q \in S$  let  $Q_i := Q \cap S_i$ .
  - b) For star  $Q$  in  $S$  let  $I_Q := \{i : |Q_i| = 1\}$  (i.e.  $Q_i$  is an edge) and let  $J_Q = \{i : |Q_i| > 1\}$ .
  - c) Let  $i_Q$  be the number of edges of  $\tilde{G}[X_3, X_4]$  that are incident to vertices from  $\bigcup_{i \in I_Q} V(Q_i) \cap X_3$ . Let  $j_Q$  be the of edges of  $\tilde{G}[X_3, X_4]$  that are incident to vertices from  $\bigcup_{i \in J_Q} V(Q_i) \cap X_3$ .

$$i_Q = \sum_{\substack{v \in V(Q_i) \cap X_3 \\ i \in I_Q}} d_{\tilde{G}[X_3, X_4]}(v)$$

$$j_Q = \sum_{\substack{v \in V(Q_i) \cap X_3 \\ i \in J_Q}} d_{\tilde{G}[X_3, X_4]}(v)$$

5. In parallel, for every star  $Q \in S$ :
  - a) If  $i_Q > j_Q$  then delete from  $Q$  all the edges that belong to  $\bigcup_{i \in J_Q} Q_i$ . In addition, select  $k \in I_Q$  such that  $k = \max I_Q$  and delete from  $Q$  all stars  $Q_i$  for which  $i \neq k$ . As a result  $Q$  is the edge  $Q_k$ .
  - b) If  $i_Q \leq j_Q$  delete from  $Q$  all the edges that belong to  $\bigcup_{i \in I_Q} Q_i$ .

Let  $S'$  denote the star forest after these operations.
6. Construct the following auxiliary multi-graph  $\hat{G} = (X_1, X_2', \hat{E})$  with  $X_2' = \{N_{X_2}(Q') : Q' \in S'\}$  and the set of edges  $\hat{E} = \{\{u, v\} \in \tilde{G} : u \in X_1 \text{ and } v \in N_{X_2}(Q') \text{ for some } Q' \in S'\}$ .
7. Find a  $1/2$ -substantial matching  $M'$  in  $\hat{G}$ .
8. Extend (in a unique way) every edge of  $M'$  to a path  $P$  of length three in the graph using an edge of matching  $M$  and an edge of a star in  $S'$ .  $\mathcal{P}$  is the set of all paths  $P$  obtained from  $M'$ .

---

We will need some more notation in the next two lemmas. Recall that  $S$  is a set of stars with centers in  $X_4$ . Let  $F_4$  be the set of vertices of stars from  $S$  which are in  $X_4$ ,  $F_3$  the set of vertices of stars in  $S$  which are in  $X_3$ . In addition let  $F_2 := N(F_3) \cap X_2$ . Similarly, by considering  $S'$ , we define  $F'_4, F'_3$ , and  $F'_2$ . The following lemma estimates the number of edges that we may lose while concentrating on the stars only.

**Lemma 3.** 1.  $e_{\tilde{G}}(F'_3, X_4) \geq \frac{1}{32} e_{\tilde{G}}(X_3, X_4)$   
2.  $e_{\tilde{G}}(X_1, F'_2) \geq \frac{1}{32} e_{\tilde{G}}(X_1, X_2)$ .

*Proof (Sketch).* By Fact 5, the proof of both parts is the same, and it follows from a property of the spanners  $S_i$  (see Part 1. of Definition 3).

**Lemma 4.** Let  $K = K(4)$  be given in Lemma 2 and let  $\xi = \frac{1}{2^{56K}}$ . The set of paths  $\mathcal{P}$  found by PATHSIN4LGRAPH is  $\xi$ -path-substantial in  $\tilde{G}$ . The algorithm PATHSIN4LGRAPH runs in  $O(\log^3 V(\tilde{G}))$  steps.

*Proof (Sketch).* First note that the time complexity comes from finding spanners in all the blocks (in parallel) in step (3) and by Lemma 2 is  $O(\log^3 n)$ . Let  $\mathcal{P}$  be the set of  $(M, 3)$ -paths found by the procedure PATHSIN4LGRAPH. In view of Fact 5, to prove that  $\mathcal{P}$  is  $\xi$ -path-substantial, we need to show that either  $\xi$ -fraction of edges in  $\tilde{G}[X_1, X_2]$  is incident to edges of paths from  $\mathcal{P}$  or  $\xi$ -fraction of edges in  $\tilde{G}[X_3, X_4]$  is incident to edges of paths from  $\mathcal{P}$ . For that we consider a matching  $M'$  in  $\hat{G}$  found in Step (7) and the set  $W$  of vertices in  $X_2$  that are in  $M'$ -saturated supervertices and are not saturated by  $M'$  themselves. Then we consider two cases based on the number of edges incident to  $W$  with respect to  $e_{\tilde{G}}(X_1, X_2)$ . If it is relatively small then the fact that  $M'$  is  $\frac{1}{2}$ -substantial combined with Lemma 3 implies that  $\xi = \frac{1}{128}$ . The second case is a bit more complicated. We move our attention to the layers  $X_3$  and  $X_4$ . Incorporating the properties of the star forest  $S'$  and the upper bound on the degree of spanners, we show that at least  $\frac{1}{256}K$ -fraction of edges in  $E_{\tilde{G}}(X_3, X_4)$  is incident to centers of stars saturated by paths from  $\mathcal{P}$ . The two cases yield the Lemma with  $\xi = \frac{1}{256}K$ .

### 3.2 Reduction, Translation, and Modification

As indicated in previous sections our main algorithm can be summarized as follows. First compute a maximal matching  $M$ , next iterate  $O(\log n)$  times the following steps: (1) Reduce graph  $G$  and matching  $M$  to a layered graph  $\tilde{G}$ , (2) Find a set  $\bar{P}$  of disjoint  $(M, 3)$ -paths in  $\tilde{G}$  using PATHSIN4LGRAPH, (3) Translate paths from  $\bar{P}$  to paths  $P$  in  $G$  maintaining the substantial property, (4) Modify  $G$  and  $\tilde{G}$  with respect  $P$ . In this section, we present procedures:

1. REDUCE which reduces  $G$  and  $M$  to a 4L-graph  $\tilde{G}$ ;
2. TRANSLATE which translates  $\bar{P}$  to a set of paths  $P$  in  $G$ ;
3. MODIFY which modifies  $\tilde{G}$  with respect to  $P$ .

Procedures REDUCE and MODIFY are similar to procedures in [2], TRANSLATE differs from the corresponding procedure in [2] only by small details.

---

#### Procedure Reduce

---

Input: Graph  $G$  and a maximal matching  $M$  in  $G$ .

Output: 4L-graph  $\tilde{G}$ .

1. For  $e \in E(G)$  (in parallel) check if  $e$  is contained in at least one  $(M, 3)$ -path. If it is not then delete  $e$ . In particular, all edges from  $E(G) \setminus M$  which have both endpoints in  $M$  are deleted.
2. For every edge  $m = \{m_1, m_2\} \in M$ , with  $ID(m_1) < ID(m_2)$ , let  $T_m$  be the set of vertices in  $V$  such that every vertex from  $T_m$  is adjacent to  $m_1$  and  $m_2$ . Partition  $T_m$  into two groups  $T_{m,1}$  and  $T_{m,2}$  so that  $||T_{m,1}| - |T_{m,2}|| \leq 1$ . For every vertex  $t \in T_{m,1}$  delete the edge  $\{t, m_2\}$  from the graph, for every vertex  $t \in T_{m,2}$  delete  $\{t, m_1\}$ . As a result, the "new" graph  $G'$  does not have triangles based on edges from  $M$ . From now on we will operate on  $G'$ .

3. For every edge  $m = \{m_1, m_2\} \in M$ , with  $ID(m_1) < ID(m_2)$ , if  $e \in E(G')$  and  $e = \{v, m_1\}$  for some  $v \neq m_2$  then orient  $e$  from  $v$  to  $m_1$ , that is delete  $e$  and add arc  $(v, m_1)$ . If  $e \in E(G')$  and  $e = \{v, m_2\}$  for some  $v \neq m_1$  then orient  $e$  from  $m_2$  to  $v$ , that is delete  $e$  and add arc  $(m_2, v)$ .
4. For every vertex  $v \in G \setminus V(M)$ , split  $v$  into two siblings  $v^-$  and  $v^+$ , where  $v^-$  inherits all the arcs that start in  $v$ ,  $v^+$  inherits arcs that end in  $v$ . Finally, ignore the orientation on the edges. As a result we obtain a 4L-graph  $\bar{G} = (V^-, V_1(M), V_2(M), V^+)$ , where  $V^- = \{v^-, v \in V(G) \setminus V(M)\}$ ,  $V^+ = \{v^+, v \in V(G) \setminus V(M)\}$ , and  $V_1(M)$ ,  $V_2(M)$  are corresponding endpoints of  $M$  that is if  $m = \{m_1, m_2\} \in M$  and  $ID(m_1) < ID(m_2)$  then  $m_1 \in V_1(M)$ ,  $m_2 \in V_2(M)$ .

---

The above procedure obtains  $\bar{G}$  by first deleting all triangles which are based on edges from  $M$  and then using orientation to split  $M$ -unsaturated vertices into  $V^-$  and  $V^+$  (see [2] for details). There are two main properties of REDUCE that are useful for our analysis.

**Lemma 5.** *Let  $\bar{m}_3$  denote the number of  $(M, 3)$ -paths in  $\bar{G}$ ,  $m'_3$  the number of  $(M, 3)$ -paths in  $G'$ , and  $m_3$  denote the number of  $(M, 3)$ -paths in  $G$ . Then*

1.  $m'_3 = \bar{m}_3$ ,
2.  $m_3/4 \leq m'_3$ .

**Fact 6.** *If  $v_1^- m_1 m_2 v_2^+$  is an  $(M, 3)$ -path in  $\bar{G}$  then  $v_1 m_1 m_2 v_2$  is  $(M, 3)$ -path in  $G$ .*

Our next procedure modifies  $G$  with respect to the set of  $(M, 3)$ -paths  $\mathcal{P}$ . The procedure deletes all edges which are incident to paths from  $\mathcal{P}$ . As a result, it is possible that some edges present in the modified graph  $G'$  will not belong to any  $(M, 3)$ -path of  $G'$ . These edges are subsequently deleted in step (1) of REDUCE.

---

### Procedure Modify

---

Input: Graph  $G$  and a set  $\mathcal{P}$  of  $(M, 3)$ - paths in  $G$ .

Output: Modified  $G'$ .

1. For any edge  $e \in \bigcup_{P \in \mathcal{P}} E(P)$  in parallel: delete edges that are incident to  $e$
2. Delete all edges from  $\bigcup_{P \in \mathcal{P}} E(P)$

---

Finally, in the last procedure of this section, we show how to translate a set of paths  $\bar{\mathcal{P}}$  in  $\bar{G}$  to a set of paths  $\mathcal{P}$  in  $G$ . Recall that  $G'$  denotes the subgraph of  $G$  obtained by destroying triangles that contain edges from  $M$ , in step (2) of REDUCE.

---

### Procedure Translate

---

Input: Set  $\bar{\mathcal{P}}$  of independent  $(M, 3)$ -paths in  $\bar{G}$ .

Output: Set  $\mathcal{P}$  of independent  $(M, 3)$ -paths in  $G$ .

1. Identify vertices  $v^-$  and  $v^+$  that correspond to one vertex  $v \in V(G)$ . As a result we obtain from  $\bar{\mathcal{P}}$  cycles and paths in graph  $G$ . These paths and cycles have lengths of the form  $3i$  where  $i > 1$  and are built up from  $(M, 3)$ -paths.
2. Treat  $(M, 3)$ -paths as edges between endpoints of these paths. Now the problem of selecting a set of independent  $(M, 3)$ -paths in  $G$  is reduced to the one of finding a substantial set of independent edges. Invoke an algorithm from [4] to obtain set  $\mathcal{P}$  of independent (in  $G$ )  $(M, 3)$ -paths. The set  $\mathcal{P}$  has the following property: If the number of  $(M, 3)$ -paths in  $G'$  which are incident to edges of  $\bigcup_{P \in \bar{\mathcal{P}}} E(P)$  is at least  $\alpha m'_3$  then at least  $\alpha m'_3/4$ ,  $(M, 3)$ -paths in  $G'$  are incident to edges of  $\bigcup_{P \in \mathcal{P}} E(P)$ .

---

Note that the resulting set  $\mathcal{P}$  is a set of  $(M, 3)$ -paths which are independent in  $G$ . In addition the set is path-substantial with a constant specified in the next lemma.

**Lemma 6.** *If a set of paths  $\bar{\mathcal{P}}$  is  $\xi$ -path-substantial in  $\bar{G}$  then the set of paths  $\mathcal{P}$  obtained by procedure TRANSLATE is  $\xi/16$ -path-substantial in  $G$ .*

*Proof.* The proof follows from Lemma 5 and Step (2) of TRANSLATE.

### 3.3 Main Procedure

Now we can summarize the discussion in our main algorithm. We denote by  $A \div B$  the symmetric difference between  $A$  and  $B$ .

---

#### Procedure Matching

---

Input: Graph  $G$ .

Output: Matching  $M^*$  such that  $|M^*| \geq \frac{2}{3}\beta(G)$ .

1. Find a maximal matching  $M$  using the procedure MATCH from [4].
  2. Let  $\mathcal{P}_I := \emptyset$  and  $M^* := M$ .
  3. Iterate  $O(\log n)$  times:
    - a) Invoke REDUCE to obtain a 4L-graph  $\bar{G}$ .
    - b) Invoke PATHSIN4LGRAPH to obtain  $\bar{\mathcal{P}}$ .
    - c) Use TRANSLATE to obtain  $\mathcal{P}$ .
    - d) Use MODIFY to modify  $\bar{G}$  with respect to  $\mathcal{P}$ .
    - e)  $\mathcal{P}_I := \mathcal{P}_I \cup \mathcal{P}$ .
  4.  $M^* := M^* \div (\bigcup_{P \in \mathcal{P}_I} E(P))$ .
- 

*Proof (of Theorem 3).* First we establish the time complexity. We use  $O(\log^4 n)$  rounds in the first step, to find  $M$  using the algorithm from [4]. In each iteration, the main time complexity comes from PATHSIN4LGRAPH which, by Lemma 4 is  $O(\log^3 V(\bar{G})) = O(\log^3 n)$ . Since we have  $O(\log n)$  iterations, the number of steps is  $O(\log^4 n)$ .

By Lemma 4, the set of paths  $\bar{\mathcal{P}}$  found in each iteration is  $\frac{1}{256K}$ -path-substantial in  $\bar{G}$ . Therefore, by Lemma 6,  $\mathcal{P}$  is  $\frac{1}{4096K}$ -path-substantial in  $G$  and so in step (3d) a constant fraction of  $(M, 3)$ -paths will be deleted from  $G$ . Since the number of  $(M, 3)$ -paths in  $G$  is  $O(n^4)$ , there will be no  $(M, 3)$ -paths in  $G$  after  $O(\log n)$  iterations. Consequently, after step (3),  $\mathcal{P}_I$  is a maximal set of  $(M, 3)$ -paths. Thus, after exchanging edges in step (4), we obtain matching  $M^*$  such that there is no  $(M^*, 3)$ -path in  $G$ . Therefore, by Claim 4,  $|M^*| \geq \frac{2}{3}\beta(G)$ .

*Final Remarks.* An efficient randomized algorithm for approximating the maximum matching can be obtained by invoking a randomized MIS procedure ([7]) in an auxiliary graph where the vertex set consists of augmenting paths and two vertices are connected if the corresponding paths share a vertex in the original graph.

The augmenting paths technique can only be applied to an unweighted version of the problem; to approximate a maximum weighted matching a completely different approach must be taken.

## References

1. Awerbuch, B., Goldberg, A.V., Luby, M., Plotkin, S.: Network decomposition and locality in distributed computing. Proc. of the 30th Symposium on Foundations of Computer Science (FOCS 1989) 364–369
2. Czygrinow, A., Hańćkowiak, M., Szymańska, E.: Distributed algorithm for approximating the maximum matching. Discrete Applied Math. published online (March 19, 2004)
3. Fischer, T., Goldberg, A.V., Haglin, D. J., Plotkin, S.: Approximating matchings in parallel. Information Processing Letters **46** (1993) 115–118
4. Hańćkowiak, M., Karoński, M., Panconesi, A.: On the distributed complexity of computing maximal matchings. SIAM J. Discrete Math. Vol.15, No.1, (2001) 41–57
5. Lovász, L., Plummer, M.: Matching Theory. Elsevier, Amsterdam and New York (1986)
6. Linial, N.: Locality in distributed graph algorithms. SIAM Journal on Computing **21**(1) (1992) 193–201
7. Luby, M.: A Simple Parallel Algorithm for the Maximal Independent Set Problem. SIAM J. on Computing **15**(4) (1986) 254–278.



# Extreme Points Under Random Noise<sup>\*</sup>

Valentina Damerow and Christian Sohler

Heinz Nixdorf Institute, Computer Science Department, University of Paderborn,  
D-33102 Paderborn, Germany  
{vio,csohler}@upb.de

**Abstract.** Given a point set  $\mathcal{P} = \{p_1, \dots, p_n\}$  in the  $d$ -dimensional unit hypercube, we give upper bounds on the maximal expected number of extreme points when each point  $p_i$  is perturbed by small random noise chosen independently for each point from the same noise distribution  $\Delta$ . Our results are parametrized by the variance of the noise distribution. For large variance we essentially consider the average case for distribution  $\Delta$  while for variance 0 we consider the worst case. Hence our results give upper bounds on the number of extreme points where our input distributions range from average case to worst case.

Our main contribution is a rather general lemma that can be used to obtain upper bounds on the expected number of extreme points for a large class of noise distributions. We then apply this lemma to obtain explicit bounds for random noise coming from the Gaussian normal distribution of variance  $\sigma^2$  and the uniform distribution in a hypercube of side length  $\epsilon$ . For these noise distributions we show upper bounds of  $\mathcal{O}((\frac{1}{\sigma})^d \cdot \log^{3/2 \cdot d-1} n)$  and  $\mathcal{O}((\frac{n \log n}{\epsilon})^{d/(d+1)})$ , respectively. Besides its theoretical motivation our model is also motivated by the observation that in many applications of convex hull algorithms the input data is inherently noisy, e.g. when the data comes from physical measurement or imprecise arithmetic is used.

**Keywords:** Convex Hull, Average Complexity, Smoothed Analysis

## 1 Introduction

The convex hull of a point set in the  $d$ -dimensional Euclidean space is one of the fundamental combinatorial structures in *computational geometry*. Many of its properties have been studied extensively in the last decades. In this paper we are interested in the number of vertices of the convex hull of a random point set, sometimes referred to as the *number of extreme points* of the point set. It is known since more than 40 years that the number of extreme points of a point set drawn uniformly at random from the (unit) hypercube is  $\mathcal{O}(\log^{d-1}(n))$ . The number of extreme points has also been studied for many other

---

<sup>\*</sup> Research is partially supported by DFG grant 872/8-2, by the DFG Research Training Group GK-693 of the Paderborn Institute for Scientific Computation (PaSCo), and by the EU within the 6th Framework Programme under contract 001907 "Dynamically Evolving, Large Scale Information Systems" (DELIS). Part of this research has been presented at the 20th European Workshop on Computational Geometry (EWCG'04).

input distributions, e.g. for the Gaussian normal distribution. Since the expectation is the average over all inputs weighted by their probabilities one also speaks of *average case number of extreme points* (and average case complexity, in general).

Research in the average case complexity of structures and algorithms is motivated by the assumption that a typical input in an application behaves like an input from the distribution considered for the average case analysis. One weakness of average case analysis is that in many applications the inputs have special properties. Compared to this, the distributions typically considered in the context of average case analysis often provide a too optimistic complexity measure.

While average case analysis may be too optimistic we also know that worst case complexity can be too pessimistic. For example, the simplex algorithm performs extraordinary well in practice but for many variants it is known that their worst case running time can be exponentially.

Since the typical complexity of a structure or algorithm lies usually somewhere between its average case complexity and its worst case complexity, there is need for a complexity measure that bridges the gap between average case and worst case. A hybrid between average case and worst case analysis called *smoothed analysis* provides such a bridge. The idea of smoothed analysis is to weaken the worst case complexity by adding small random noise to any input instance. The *smoothed complexity* of a problem is then the worst case expected complexity, where the expectation is with respect to the random noise.

In this paper, the input is a point set and we consider the number of extreme points. We add to each point a small random vector from a certain noise distribution (typically, Gaussian noise or noise drawn uniformly from a small hypercube). If the noise distribution is fixed then any point set becomes a distribution of points and the *smoothed number of extreme points* (for this noise distribution) is the maximum expected number of extreme points over all distributions. The noise is parametrized by its variance and so smoothed analysis provides an analysis for a wide variety of distributions ranging from the average case (for large variance) to the worst case (variance 0).

Besides its appealing theoretical motivation the use of smoothed analysis is particularly well motivated in the context of computational geometry. In many applications of computational geometry and convex hull computation the input data comes from physical measurements and has thus a certain error because measuring devices have limited accuracy. A standard assumption in physics is that this error is distributed according to the Gaussian normal distribution. Thus we can use smoothed analysis with Gaussian error to model inputs coming from physical measurements. Besides the assumption that physical measurements are not precise and that the error is distributed according to the Gaussian normal distribution smoothed analysis provides us a *worst case* analysis for this class of inputs.

Another interesting motivation for smoothed analysis in the context of computational geometry is that sometimes fixed precision arithmetic is used during computations. We consider the case that the input points are computed by fixed precision arithmetic and we model this scenario by the assumption that every point is distributed uniformly in a hypercube around its 'real' position.

## 1.1 Our Contribution

In this paper we consider the smoothed number of extreme points for random noise from a large class of noise distributions. We consider an arbitrary input point set  $\mathcal{P}$  in the unit hypercube. To each point random noise from a noise distribution  $\Delta$  is added. We assume that  $\Delta$  is the  $d$ -fold product of a one dimensional distribution  $\Delta_1$  with mean 0. This assumption is satisfied for many natural distributions including the uniform distribution in a hypercube or the Gaussian normal distribution.

Our main technical contribution is a lemma that provides an upper bound on the number of extreme points for any noise distribution satisfying the above conditions. The bound itself depends on the variation of the local density of the distribution and two parameters that must be optimized by hand. We apply this lemma to noise from the Gaussian normal distribution and from a cube with side length  $\epsilon$ . We obtain the following results (where we consider the dimension  $d$  to be a constant):

- If  $\Delta$  is the  $d$ -fold product of a one dimensional standard Gaussian normal distribution of variance  $\sigma^2$  we show that the smoothed number of extreme points is

$$\mathcal{O} \left( \left( \frac{1}{\sigma} \right)^d \cdot \log^{3/2 \cdot d - 1} n \right) .$$

- If  $\Delta$  is a uniform distribution from a hypercube of side length  $\epsilon$  we prove that the smoothed number of extreme points is

$$\mathcal{O} \left( \left( \frac{n \log n}{\epsilon} \right)^{d/(d+1)} \right) .$$

## 1.2 Related Work

Several authors have treated the structure of the convex hull of  $n$  random points. In 1963/64, Rényi and Sulanke [8,9] were the first to present the mean number of extreme points in the planar case for different stochastic models. They showed for  $n$  points uniformly chosen from a convex polygon with  $r$  vertices a bound of  $\mathcal{O}(r \cdot \log n)$ . This work was continued by Efron [4], Raynaud [6,7], and Carnal [2] and extended to higher dimensions. For further information we refer to the excellent book by Santaló [10].

In 1978, Bentley, Kung, Schkolnick, and Thompson [1] showed that the expected number of extreme points of  $n$  i.i.d. random points in  $d$  space is  $\mathcal{O}(\ln^{d-1} n)$  for fixed  $d$  for some general probability distributions. Har-Peled [5] gave a different proof of this result. Both proofs are based on the computation of the expected number of maximal points (cf. Section 2). This is also related to computing the number of points on a Pareto front of a point set.

The concept of smoothed analysis was introduced in 2001 by Spielman and Teng [11]. In 2003 this concept was applied to the number of changes to the combinatorial description of the smallest enclosing bounding box of a moving point set [3] where different probability distributions for the random noise were considered. Although the two dimensional case of this paper's problem is covered in the bounding box paper [3],

the result was achieved by reduction to so-called left-to-right maxima in a sequence of numbers. A similar reduction applied in higher dimensions could only reduce the  $d$ -dimensional problem to a (different)  $(d-1)$ -dimensional problem. Hence new techniques are necessary to attack the problem in higher dimensions.

## 2 Preliminaries

For a point set  $\mathcal{P} := \{p_1, \dots, p_n\}$  in  $\mathbb{R}^d$  let  $\mathcal{V}(\mathcal{P})$  denote the number of *extreme points* (i.e. the number of vertices of the convex hull) of  $\mathcal{P}$ . Further let  $\Delta$  denote a probability distribution over the  $\mathbb{R}^d$ . Throughout the paper we only consider probability distributions  $\Delta$  that are the  $d$ -fold product of a one dimensional probability distribution  $\Delta_1$  with mean 0 (e.g. uniform distribution in the unit hypercube, Gaussian normal distribution, etc.). Let  $\varphi : \mathbb{R} \rightarrow \mathbb{R}$  denote the probability density function of  $\Delta_1$  and let  $\Phi(\xi^{(i)}) := \Pr[p^{(i)} \leq \xi^{(i)}] = \int_{-\infty}^{\xi^{(i)}} \varphi(x) dx$  denote its probability distribution function. Then the  $d$ -dimensional probability distribution of  $\Delta$  is given by

$$\Pr[p^{(1)} \leq \xi^{(1)}, \dots, p^{(d)} \leq \xi^{(d)}] = \int_{-\infty}^{\xi^{(d)}} \dots \int_{-\infty}^{\xi^{(1)}} \varphi(x^{(1)}) \dots \varphi(x^{(d)}) dx^{(1)} \dots dx^{(d)}.$$

For a given point set  $\mathcal{P} = \{p_1, \dots, p_n\}$  in the unit  $d$ -hypercube we denote by  $\tilde{\mathcal{P}}(\Delta) = \{\tilde{p}_1, \dots, \tilde{p}_n\}$  the distribution of points obtained by adding random noise from distribution  $\Delta$  to the points in  $\mathcal{P}$ . The distribution of point  $\tilde{p}_i$  is given by  $\tilde{p}_i = p_i + r_i$ , where  $r_i$  is a random vector chosen (independently of the other  $r_j$ 's) from  $\Delta$ . The probability distribution of  $\tilde{\mathcal{P}}(\Delta)$  is then given by combining the distributions of the  $\tilde{p}_i$ . We will parametrize  $\Delta$  by its variance. Thus our results depend on the ratio between variance and diameter of the point set. When the noise distribution is clear from the context we will also use  $\tilde{\mathcal{P}}$  to abbreviate  $\tilde{\mathcal{P}}(\Delta)$ .

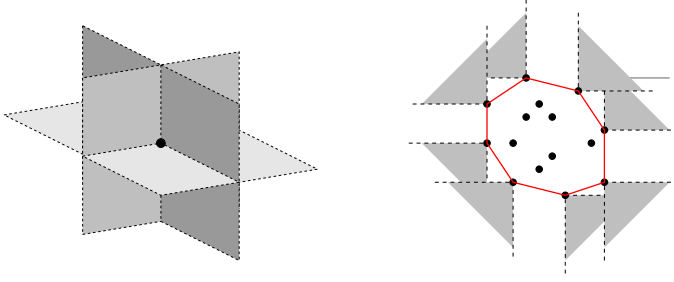
We define the *smoothed number*  $\mathcal{V}(\tilde{\mathcal{P}}(\Delta))$  of extreme points to be the maximal expected number of extreme points in a perturbed point set over all sets  $\mathcal{P}$  in the unit hypercube, i.e.  $\max_{\mathcal{P} \subset [0,1]^d} \mathbf{E}[\text{number of extreme points in } \tilde{\mathcal{P}}(\Delta)]$ .

### 2.1 Outline

Our bounds are based on the following observation: a point  $p \in \mathcal{P}$  is not extreme, if each of the  $2^d$  orthants centered at  $p$  contains at least one point. In this case, we say that  $p$  is not *maximal*. It follows immediatly that the number of maximal points is an upper bound on the number of extreme points. Therefore, we will from now on count the number of maximal points.

In Section 3 we consider *average case* for points sets of  $n$  points drawn independently from a distribution  $\Delta$  of the form described above. We show that for every such  $\Delta$  the expected number of extreme points is  $\mathcal{O}(\log^{d-1} n)$ . We remark that this result is well-known. We present it to illustrate our approach in the smoothed case.

In Section 4 we develop our main lemma and apply it to obtain upper bounds on the smoothed number of extreme points for noise coming from the Gaussian normal distribution and the uniform distribution in a hypercube.



**Fig. 1.** 'A point in three dimensional space has eight orthants.' and 'Every extreme point is also a maximal point.'

### 3 Average Case

We illustrate our approach on the well-understood case of  $n$  points in  $\mathbb{R}^d$  chosen from a probability distribution  $\Delta$  that is the  $d$ -fold product of a one dimensional probability distribution. We show how to obtain in this case an upper bound of  $\mathcal{O}(\log^{d-1} n)$  on the number of maximal points and hence on the number of extreme points.

**Theorem 1.** *Let  $\mathcal{P} = \{p_1, \dots, p_n\}$  be a set of  $n$  i.i.d. random points in  $\mathbb{R}^d$  chosen according to the probability distribution  $\Delta$ . Then the expected number of extreme points of  $\mathcal{P}$  is  $\mathcal{O}(\log^{d-1} n)$  for fixed dimension  $d$ .*

*Proof.* To prove the theorem we show that

$$\Pr[p_k \text{ is maximal}] = \mathcal{O}(\log^{d-1} n/n) . \quad (1)$$

By linearity of expectation it follows immediately that the number of extreme points is  $\mathcal{O}(\log^{d-1} n)$ . To prove (1) we consider the probability that a fixed orthant  $\mathbf{o}(p_k)$  is empty. Using a standard union bound we get

$$\Pr[p_k \text{ is maximal}] \leq 2^d \cdot \Pr[\mathbf{o}(p_k) \text{ is empty}] .$$

W.l.o.g. we now fix orthant  $\mathbf{o}(p_k) := \prod_{i=1}^d (-\infty, p_k^{(i)}]$ . We can write the probability that  $\mathbf{o}(p_k)$  is empty as an integral. For reasons of better readability we establish first the 2-dimensional integral.

Consider  $p_k$  having the coordinates  $(x, y)$ . The probability for any other point  $p_j \in \mathcal{P} - \{p_k\}$  to be not in  $\mathbf{o}(p_k)$  is then equal to the probability that the point  $p_j$  lies in one of the other orthants of  $p_k$ . E.g. for the first orthant  $[p_k^{(1)}, \infty) \times [p_k^{(2)}, \infty) =: \mathbf{o}_1$  this probability is

$$\Pr[p_j \in \mathbf{o}_1] = \Pr[p_j^{(1)} \geq x] \cdot \Pr[p_j^{(2)} \geq y] = (1 - \Phi(x)) \cdot (1 - \Phi(y)) .$$

The probability for the remaining 2 orthants can be expressed similarly yielding for the probability

$$\begin{aligned} \Pr[p_j \notin \mathbf{o}(p_k)] &= (1 - \Phi(x)) \cdot \Phi(y) + (1 - \Phi(x)) \cdot (1 - \Phi(y)) + \Phi(x) \cdot (1 - \Phi(y)) \\ &= 1 - \Phi(x) \cdot \Phi(y) . \end{aligned}$$

Since there are  $n - 1$  other points in  $\mathcal{P}$  the probability that  $\mathbf{o}(p_k)$  is empty is exactly

$$\int_{\mathbb{R}^2} \varphi(x) \cdot \varphi(y) \cdot \underbrace{(1 - \Phi(x) \cdot \Phi(y))^{n-1}}_{=: z = z(x)} \, dx dy . \quad (2)$$

We solve this integral by the indicated substitution where  $dz = -\varphi(x) \cdot \Phi(y) \cdot dx$ . We get

$$\int_{\mathbb{R}} \int_{1-\Phi(y)}^1 \frac{\varphi(y)}{\Phi(y)} \cdot z^{n-1} \, dz dy = \frac{1}{n} \int_{\mathbb{R}} \frac{\varphi(y)}{\Phi(y)} \cdot (1 - \underbrace{(1 - \Phi(y))^n}_{=: z}) dy .$$

Another substitution yields, where  $dz = -\varphi(y) \cdot dy$

$$\frac{1}{n} \int_0^1 \frac{1 - z^n}{1 - z} \, dz = \frac{1}{n} \sum_{i=0}^{n-1} \int_0^1 z^i \, dz = \frac{1}{n} \sum_{i=1}^n \frac{1}{i} = \frac{\log n + \mathcal{O}(1)}{n} .$$

Since there are 4 orthants and  $n$  points it follows that the expected number of maximal points in the planar case is  $\mathcal{O}(\log n)$ .

The integral (2) for the  $d$ -dimensional case is of the following form and can be solved by repeated substitutions in a similar fashion:

$$\begin{aligned} & \int_{\mathbb{R}^d} \varphi(x^{(1)}) \cdots \varphi(x^{(d)}) \cdot \left( \sum_{\mathcal{I} \subset [d]} \prod_{i \in \mathcal{I}} \Phi(x^{(i)}) \prod_{i \notin \mathcal{I}} (1 - \Phi(x^{(i)})) \right)^{n-1} dx^{(1)} \cdots dx^{(d)} \\ &= \frac{1}{n} \sum_{i_1=1}^n \frac{1}{i_1} \sum_{i_2=1}^{i_1} \frac{1}{i_2} \cdots \sum_{i_{d-1}=1}^{i_{d-2}} \frac{1}{i_{d-1}} = \mathcal{O} \left( \frac{\log^{d-1} n}{n} \right) . \end{aligned}$$

The expected number of extreme points in  $d$  dimensions is then  $\mathcal{O}(\log^{d-1} n)$ .  $\square$

## 4 Smoothed Case

We now want to apply the same approach to obtain an upper bound on the smoothed number of extreme points. We consider a perturbed point  $\tilde{p}_k = p_k + r_k$ , where  $p_k = (p_k^{(1)}, \dots, p_k^{(d)}) \in [0, 1]^d$  and  $r_k$  a random  $d$ -vector chosen from a probability distribution which is the  $d$ -fold product a one dimensional probability distribution of mean 0. Let again  $\varphi : \mathbb{R} \rightarrow \mathbb{R}$  be the probability density of that distribution.

Again we want to bound the probability that a fixed orthant of a perturbed point is empty. We make the observation that perturbed point  $\tilde{p}_k$  is a random vector from a probability distribution of mean  $p_k$  with one dimensional density function  $\varphi(x - p_k)$  and distribution function  $\Phi(x - p_k)$  for its components. In other words, perturbed point  $\tilde{p}_k$  has a slightly shifted density and distribution function. Then

$$\begin{aligned} \Pr[\mathbf{o}(\tilde{p}_k) \text{ is empty}] &= \int_{\mathbb{R}^d} \varphi(x^{(1)} - p_k^{(1)}) \cdots \varphi(x^{(d)} - p_k^{(d)}) \cdot \\ &\prod_{j \neq k} \sum_{\mathcal{I} \subset [d]} \prod_{i \in \mathcal{I}} \Phi(x^{(i)} - p_j^{(i)}) \prod_{i \notin \mathcal{I}} (1 - \Phi(x^{(i)} - p_j^{(i)})) \, dx^{(1)} \cdots dx^{(d)} . \end{aligned}$$

The main idea is now to subdivide the unit hypercube into  $m = 1/\delta^d$  smaller axis-aligned hypercubes of side length  $\delta$ . Then we subdivide  $\mathcal{P}$  into sets  $\mathcal{C}_1, \dots, \mathcal{C}_m$  where  $\mathcal{C}_\ell$  is the subset of  $\mathcal{P}$  that is located (before the perturbation) in the  $\ell$ -th small hypercube (we assume some ordering among the small hypercubes). Now we can calculate the expected number  $\mathcal{D}(\mathcal{C}_\ell)$  of maximal points for the sets  $\mathcal{C}_\ell$  and use

$$\mathcal{V}(\tilde{\mathcal{P}}) \leq \sum_{\ell=1}^m \mathcal{V}(\tilde{\mathcal{C}}_\ell) \leq \sum_{\ell=1}^m \mathcal{D}(\tilde{\mathcal{C}}_\ell) \quad (3)$$

to obtain an upper bound on the expected number of extreme points in  $\tilde{\mathcal{P}}$ . The advantage of this approach is that for small enough  $\delta$  the points in a single small hypercube behave almost as in the unperturbed random case.

We now want to compute the expected number of extreme points for the sets  $\mathcal{C}_\ell$ . We assume w.l.o.g. that set  $\mathcal{C}_\ell$  is in the hypercube  $[0, \delta]^d$  and that  $\mathcal{C}_\ell$  is of magnitude  $c_\ell$ . Let  $\bar{\delta} = (\delta, \dots, \delta)$  and  $\bar{0} = (0, \dots, 0)$ . Again we aim to find an upper bound on the probability that for a point  $p_k \in \mathcal{C}_\ell$  the orthant  $\mathbf{o}(\tilde{p}_k)$  is empty. This probability is maximized if  $p_k = \bar{0}$  and  $p_j = \bar{\delta}$  for every  $p_j \in \mathcal{C}_\ell$ ,  $p_j \neq p_k$ . Hence, we get

$$\Pr[\mathbf{o}(\tilde{p}_k) \text{ is empty}] \leq \int_{\mathbb{R}^d} \varphi(x^{(1)}) \cdots \varphi(x^{(d)}) \cdot \quad (4)$$

$$\left( \sum_{\mathcal{I} \subset [d]} \prod_{i \in \mathcal{I}} \Phi(x^{(i)} - \delta) \prod_{i \notin \mathcal{I}} (1 - \Phi(x^{(i)} - \delta)) \right)^{c_\ell - 1} dx^{(1)} \cdots dx^{(d)} \quad (5)$$

In a next step we will expand the product of density functions in (4) by a multiplication in the following way

$$\varphi(x^{(1)}) \cdots \varphi(x^{(d)}) = \varphi(x^{(1)} - \delta) \cdots \varphi(x^{(d)} - \delta) \cdot \frac{\varphi(x^{(1)})}{\varphi(x^{(1)} - \delta)} \cdots \frac{\varphi(x^{(d)})}{\varphi(x^{(d)} - \delta)}.$$

Where the ratios are bounded we can extract them from the integral and solve the remaining integral as in the average case analysis. We summarize this technique in the following crucial lemma:

**Main Lemma.** *Let  $\varphi$  be the density of the probability distribution of the random noise. For positive parameters  $\delta$  and  $r$  define the set  $\mathcal{Z}_{\delta,r}^\varphi := \left\{ x \in \mathbb{R}^d ; \frac{\varphi(x)}{\varphi(x-\delta)} > r \right\}$ . Let  $\mathcal{Z}$  be the probability of set  $\mathcal{Z}_{\delta,r}^\varphi$ , i.e.*

$$\mathcal{Z} := \sum_{i=0}^{d-1} \binom{d}{i} \underbrace{\int_{\mathcal{Z}_{\delta,r}^\varphi} \cdots \int_{\mathcal{Z}_{\delta,r}^\varphi}}_{d-i} \underbrace{\int_{\mathbb{R} - \mathcal{Z}_{\delta,r}^\varphi} \cdots \int_{\mathbb{R} - \mathcal{Z}_{\delta,r}^\varphi}}_i \varphi(x^{(1)}) \cdots \varphi(x^{(d)}) dx^{(1)} \cdots dx^{(d)}.$$

*The smoothed number of extreme points is then*

$$\mathcal{V}(\tilde{\mathcal{P}}) \leq 2^d \cdot \left( r^d \cdot \left( \frac{1}{\delta} \right)^d \cdot \log^{d-1} n + n \cdot \mathcal{Z} \right).$$

*Proof.* For better readability we first look only at the 2-dimensionisnal case. We consider again the points  $\mathcal{C}_\ell \subseteq \mathcal{P}$  lying in the small hypercube  $[0, \delta]^2$ . After some standard simplifications the integral (4,5) is of the form

$$\int_{\mathbb{R}^2} \varphi(x) \cdot \varphi(y) \cdot (1 - \Phi(x - \delta) \cdot \Phi(y - \delta))^{c_\ell - 1} dx dy . \quad (6)$$

Recall that  $\mathcal{Z}_{\delta, r}^\varphi := \left\{ x \in \mathbb{R} ; \frac{\varphi(x)}{\varphi(x - \delta)} > r \right\}$  is the subset of  $\mathbb{R}$  that contains all elements  $x$  for which the ratio  $\varphi(x)/\varphi(x - \delta)$  is larger than  $r$ . Now where the ratio of densities is bounded we expand the product of the density functions in the described way and pull the factor  $r$  in front of the integral. We obtain an upper bound on integral (6) of

$$r^2 \int_{\mathbb{R} - \mathcal{Z}_{\delta, r}^\varphi} \int_{\mathbb{R} - \mathcal{Z}_{\delta, r}^\varphi} \varphi(x - \delta) \cdot \varphi(y - \delta) \cdot (1 - \Phi(x - \delta) \cdot \Phi(y - \delta))^{c_\ell - 1} dx dy \quad (7)$$

$$+ \int_{\mathcal{Z}_{\delta, r}^\varphi} \int_{\mathbb{R} - \mathcal{Z}_{\delta, r}^\varphi} \varphi(x) \cdot \varphi(y) \cdot (1 - \Phi(x - \delta) \cdot \Phi(y - \delta))^{c_\ell - 1} dx dy \quad (8)$$

$$+ \int_{\mathbb{R} - \mathcal{Z}_{\delta, r}^\varphi} \int_{\mathcal{Z}_{\delta, r}^\varphi} \varphi(x) \cdot \varphi(y) \cdot (1 - \Phi(x - \delta) \cdot \Phi(y - \delta))^{c_\ell - 1} dx dy \quad (9)$$

$$+ \int_{\mathcal{Z}_{\delta, r}^\varphi} \int_{\mathcal{Z}_{\delta, r}^\varphi} \varphi(x) \cdot \varphi(y) \cdot (1 - \Phi(x - \delta) \cdot \Phi(y - \delta))^{c_\ell - 1} dx dy \quad (10)$$

Totally analogously the  $d$ -dimensional integral can be split up. The first part (7) can easily be solved by applying the average case bound of theorem 1 yielding for the  $d$  dimensional case a bound of  $r^d \cdot \log^{d-1} c_\ell / c_\ell$ . For the other integrals (8, 9, 10) we bound the expression  $(\sum_{\mathcal{I} \subseteq [d]} \prod_{i \in \mathcal{I}} \Phi(x^{(i)} - \delta) \prod_{i \notin \mathcal{I}} (1 - \Phi(x^{(i)} - \delta)))^{c_\ell - 1}$  by 1 yielding for the remaining sum of integrals an upper bound of

$$\sum_{i=0}^{d-1} \binom{d}{i} \underbrace{\int_{\mathcal{Z}_{\delta, r}^\varphi} \cdots \int_{\mathcal{Z}_{\delta, r}^\varphi}}_{d-i} \underbrace{\int_{\mathbb{R} - \mathcal{Z}_{\delta, r}^\varphi} \cdots \int_{\mathbb{R} - \mathcal{Z}_{\delta, r}^\varphi}}_i \varphi(x^{(1)}) \cdots \varphi(x^{(d)}) dx^{(1)} \cdots dx^{(d)}$$

which we denote by  $\mathcal{Z}$ . Now we conclude that

$$\mathcal{D}(\mathcal{C}_\ell) \leq c_\ell \cdot 2^d \cdot r^d \cdot \frac{\log^{d-1} c_\ell}{c_\ell} + c_\ell \cdot 2^d \cdot \mathcal{Z} .$$

By (3) the main lemma follows immediately.  $\square$

In the following two subsections we want to apply the main lemma to two random noise distributions, namely we consider Gaussian normal noise and uniform noise.

#### 4.1 Normal Noise

For normally distributed noise we will show the following theorem.



**Theorem 2.** *The smoothed number of extreme points under random noise from the standard normal distribution  $N(0, \sigma)$  is*

$$\mathcal{O}\left(\left(\frac{1}{\sigma}\right)^d \cdot \log^{3/2 \cdot d-1} n\right)$$

for fixed dimension  $d$ .

*Proof.* Let  $\varphi(x) := \frac{1}{\sqrt{2\pi}\sigma} \cdot e^{-\frac{x^2}{2\sigma^2}}$  be the standard normal density function with expectation 0 and variance  $\sigma^2$ . In order to utilize the main lemma we choose  $\delta := \sigma/B$  where  $B := \sqrt{\log\left(1/\left(\sqrt[d]{1+\frac{1}{n}}-1\right)\right)}$ . For  $x \leq -2\sigma B$  it holds that

$$\frac{\varphi(x)}{\varphi(x-\delta)} = e^{-\frac{\delta}{\sigma^2} \cdot x + \frac{\delta^2}{2\sigma^2}} = e^{-\frac{x}{\sigma \cdot B} + \frac{1}{2B^2}} \leq e^2 + \frac{1}{2B^2} \leq e^3.$$

Therefore, if we choose  $r := e^3$  we have  $\mathcal{Z}_{\delta,r}^\varphi \subset (-\infty, -2\sigma B]$ . The next crucial step is to bound  $\mathcal{Z}$  given by

$$\begin{aligned} \sum_{i=0}^{d-1} \binom{d}{i} \underbrace{\int_{-\infty}^{-2\sigma B} \cdots \int_{-\infty}^{-2\sigma B}}_{d-i} \underbrace{\int_{-2\sigma B}^{\infty} \cdots \int_{-2\sigma B}^{\infty}}_i \varphi(x^{(1)}) \cdots \varphi(x^{(d)}) dx^{(1)} \cdots dx^{(d)} \\ \leq \sum_{i=0}^{d-1} \binom{d}{i} \left( \int_{-\infty}^{-2\sigma B} \varphi(x) dx \right)^{d-i}. \end{aligned} \quad (11)$$

From probability theory it is well known that we can estimate the tail of the standard normal probability distribution  $N(0, \sigma)$  in the following way: it is  $\int_{-\infty}^{-k\sigma} \varphi(x) dx \leq e^{-k^2/4}$ . Hence,

$$\int_{-\infty}^{-2\sigma B} \varphi(x) dx \leq e^{-B^2} = \sqrt[d]{1+\frac{1}{n}} - 1. \quad (12)$$

Combining (11) and (12) we get that

$$\mathcal{Z} \leq \sum_{i=0}^{d-1} \binom{d}{i} \left( \sqrt[d]{1+\frac{1}{n}} - 1 \right)^{d-i} = \left( 1 + \sqrt[d]{1+\frac{1}{n}} - 1 \right)^d - 1 = \frac{1}{n}.$$

Altogether we can apply the main lemma with the earlier chosen  $\delta$ , with  $r = e^3$  and  $\mathcal{Z} \leq 1/n$ . This gives that the smoothed number of extreme points is at most

$$2^d \cdot \left( e^{3d} \cdot \left( \frac{1}{\delta} \right)^d \cdot \log^{d-1} n + 1 \right) = \mathcal{O}\left(\left(\frac{1}{\sigma}\right)^d \cdot \log^{3/2 \cdot d-1} n\right),$$

as desired.  $\square$

## 4.2 Uniform Noise

We consider random noise that is uniformly distributed in a hypercube of side length  $2\epsilon$  centered at the origin, i.e. we have the one dimensional density function

$$\varphi(x) = \begin{cases} \frac{1}{2\epsilon} & \text{if } x \in [-\epsilon, \epsilon] \\ 0 & \text{else} \end{cases}$$

for the random noise. We show

**Theorem 3.** *The smoothed number of extreme points under uniform random noise from a hypercube of side length  $2\epsilon$  is*

$$\mathcal{O}\left(\left(\frac{n \cdot \log n}{\epsilon}\right)^{\frac{d}{d+1}}\right)$$

for fixed dimension  $d$ .

*Proof.* Again we want to utilize our main lemma. We choose  $r = 1$ , then it follows immediately that for  $0 < \delta < 2\epsilon$  we have  $\mathcal{Z}_{\delta,r}^{\varphi} = \left\{x \in \mathbb{R}; \frac{\varphi(x)}{\varphi(x-\delta)} \neq 1\right\} = \mathbb{R} - [\delta - \epsilon, \epsilon]$ . We now have to compute  $\mathcal{Z}$  which is

$$\begin{aligned} \mathcal{Z} &= \sum_{i=0}^{d-1} \binom{d}{i} \underbrace{\int_{-\epsilon}^{\delta-\epsilon} \cdots \int_{-\epsilon}^{\delta-\epsilon}}_{d-i} \underbrace{\int_{\delta-\epsilon}^{\epsilon} \cdots \int_{\delta-\epsilon}^{\epsilon}}_i \left(\frac{1}{2\epsilon}\right)^d dx^{(1)} \cdots dx^{(d)} \\ &= \sum_{i=0}^{d-1} \binom{d}{i} \cdot \left(\frac{1}{2\epsilon}\right)^d \cdot \delta^{d-i} \cdot (2\epsilon - \delta)^i = \left(\frac{1}{2\epsilon}\right)^d ((2\epsilon)^d - (2\epsilon - \delta)^d) \\ &= 1 - \left(1 - \frac{\delta}{2\epsilon}\right)^d \leq 1 - \left(1 - d \cdot \frac{\delta}{2\epsilon}\right) = d \cdot \frac{\delta}{2\epsilon}. \end{aligned}$$

With the main lemma it follows that the smoothed number of extreme points is at most

$$2^d \cdot \left(\left(\frac{1}{\delta}\right)^d \cdot \log^{d-1} n + n \cdot d \cdot \frac{\delta}{2\epsilon}\right).$$

If we choose  $\delta = \mathcal{O}\left(\epsilon \cdot \sqrt[d+1]{\frac{\log^{d-1} n}{n}}\right)$  we obtain the desired bound.  $\square$

## 5 Conclusion

We analyzed the smoothed number of extreme points, which is a measure for the complexity of the convex hull of point sets under random noise. We derived bounds for several general noise distributions and gave explicit bounds for random noise from the Gaussian normal distribution and the uniform distribution in a hypercube.

There are many interesting problems concerning the smoothed complexity of geometric objects. For example, it would be interesting to analyze the smoothed number of faces of the convex hull of a point set. It would also be interesting to obtain similar results for other geometric structures like the Delaunay triangulation or the Voronoi diagram. Finally, it would be nice to either improve our bounds or find matching lower bounds.

## References

1. BENTLEY, J. L., KUNG, H. T., SCHKOLNICK, M., AND THOMPSON, C. D. On the average number of maxima in a set of vectors. *J. Assoc. Comput. Mach.* 25 (1978), 536–543.
2. CARNAL, H. Die konvexe Hülle von  $n$  rotationssymmetrisch verteilten Punkten. *Z. Wahrscheinlichkeitsth.* 15 (1970), 168–176.
3. DAMEROW, V., MEYER AUF DER HEIDE, F., RÄCKE, H., SCHEIDELER, C., AND SOHLER, C. Smoothed motion complexity. In *Proceedings of the 11th European Symposium on Algorithms (ESA)* (2003), pp. 161–171.
4. EFFRON, B. The convex hull of a random set of points. *Biometrika* 52 (1965), 331–343.
5. HAR-PELED, S. On the expected complexity of random convex hulls. Tech. Rep. 330, School Math. Sci., Tel-Aviv Univ., Tel-Aviv, Israel, 1998.
6. RAYNAUD, H. Sur le comportement asymptotique de l’enveloppe convexe d’un nuage de points tirés au hasard dans  $R^n$ . *C. R. Acad. Sci. Paris* 261 (1965), 627–629.
7. RAYNAUD, H. Sur l’enveloppe convexe des nuages de points aléatoires dans  $R^n$ . *J. Appl. Prob.* 7 (1970), 35–48.
8. RÉNYI, A., AND SULANKE, R. Über die konvexe Hülle von  $n$  zufällig gewählten Punkten I. *Z. Wahrscheinlichkeitsth.* 2 (1963), 75–84.
9. RÉNYI, A., AND SULANKE, R. Über die konvexe Hülle von  $n$  zufällig gewählten Punkten II. *Z. Wahrscheinlichkeitsth.* 3 (1964), 138–147.
10. SANTALÓ, L. A. *Integral Geometry and Geometric Probability*. Addison-Wesley Publishing Company, 1976.
11. SPIELMAN, D., AND TENG, S. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. In *Proceedings of the 33rd ACM Symposium on Theory of Computing (STOC)* (2001), pp. 296–305.

# Fixed Parameter Algorithms for Counting and Deciding Bounded Restrictive List $H$ -Colorings\*

## (Extended Abstract)

Josep Díaz, Maria Serna, and Dimitrios M. Thilikos

Dept. de Llenguatges i Sistemes Informàtics  
Universitat Politècnica de Catalunya, Campus Nord-C6.  
E-08034, Barcelona, Spain.  
`{diaz,mjserna,sedthilk}@lsi.upc.es`

**Abstract.** We study the fixed parameter tractability of the parameterized counting and decision version of the restrictive  $H$ -coloring problem. These problems are defined by fixing the number of preimages of a subset  $C$  of the vertices in  $H$  through a partial weight assignment  $(H, C, K)$ . We consider two families of partial weight assignment the *simple* and the *plain*. For simple partial weight assignments we show an FPT algorithm for counting list  $(H, C, K)$ -colorings and faster algorithms for its decision version. For the more general class of plain partial weight assignment we give an FPT algorithm for the  $(H, C, K)$ -coloring decision problem. We introduce the concept of *compactor* and an algorithmic technique, *compactor enumeration*, that allow us to design the FPT algorithms for the counting version (and probably export the technique to other problems).

## 1 Introduction

Given two graphs  $G$  and  $H$ , a *homomorphism* from  $G$  to  $H$  is any function mapping the vertices in  $G$  to vertices in  $H$ , in such a way that the image of an edge is also an edge. In the case that  $H$  is fixed, such a homomorphism is called  $H$ -coloring of  $G$ . For a given graph  $H$ , the  *$H$ -coloring problem* asks whether there exists an  $H$ -coloring of the input graph  $G$ . The more general version in which a list of allowed colors (vertices of  $H$ ) is given for each vertex is known as the list  $H$ -coloring. The case in which the number of pre-images of the vertices in  $H$  are part of the input, is known as the *restrictive  $H$ -coloring* or the *restrictive list  $H$ -coloring*. Besides the decisional versions of these problems, we also consider the problems associated to counting the number of solutions, in particular we consider the  $\#H$ -coloring, the list  $\#H$ -coloring, the restrictive  $\#H$ -coloring, and the restrictive list  $\#H$ -coloring. The known results concerning the complexity of all the problems defined so far are summarized in Table 1.

---

\* Partially supported by the EU within FP6 under contract 001907 (DELIS). The first author was further supported by the Distinció per a la Recerca of the Generalitat de Catalunya. The second and third authors were further supported by the Spanish CICYT project TIC-2002-04498-C05-03.

**Table 1.** Complexity of the parameterized  $H$ -coloring problems

Problem	P versus NP-complete/#P-complete
list $H$ -coloring	dichotomy <sup>(1)</sup> [9]
$H$ -coloring	dichotomy <sup>(2)</sup> [12]
list $\#H$ -coloring	dichotomy <sup>(3)</sup> [3,13]
$\#H$ -coloring	dichotomy <sup>(3)</sup> [7]
restrictive list $H$ -coloring	dichotomy <sup>(3)</sup> [3]
restrictive $H$ -coloring	dichotomy <sup>(3)</sup> [3]
restrictive list $\#H$ -coloring	dichotomy <sup>(3)</sup> [3]
restrictive $\#H$ -coloring	dichotomy <sup>(3)</sup> [3]
list $(H, C, K)$ -coloring	dichotomy <sup>(3)</sup> [2,5]
$(H, C, K)$ -coloring	P: list $(H - C)$ -coloring in P NP-hard: $(H - C)$ -coloring NP-hard [2,5]
list $\#(H, C, K)$ -coloring	dichotomy <sup>(3)</sup> [2,5]
$\#(H, C, K)$ -coloring ( $K$ fixed)	P: list $\#(H - C)$ -coloring in P #P-complete: $(H, C, K)$ irreducible [2,5]

- (1)  $H$  is a bi-arc graph.
- (2)  $H$  is bipartite or contains a loop.
- (3) All the connected components of  $H$  are either complete reflexive graphs. or complete irreflexive bipartite graphs.

A parameterization for the restrictive  $H$ -coloring problem was introduced in [2], and it is denoted as  $(H, C, K)$ -coloring. The parameterized version of the *restrictive  $H$ -coloring* where  $C$  is the set of vertices with restrictions and  $K$  is a weight assignment on  $C$  defining the number of preimages.  $(H, C, K)$  is called the *partial weight assignment* that fixes the parameterized part of the problem. We studied the complexity of the  $(H, C, K)$ -coloring in [2,5] (see also [4]) and the results are summarized in Table 1.

Recall that an FPT algorithm for a parameterized problem with input size  $n$  and  $k$  as parameter is one that solves the problem in time  $O(f(k)n^{O(1)})$  where  $k$  is the parameter and  $n$  is the input size. We isolate a subclass of the problem, the *simple partial weight assignments* (see definition later), for which we were able to design linear time fixed parameter algorithms for both counting and decision version of the list  $(H, C, K)$ -problem. In the case of the  $(H, C, K)$ -coloring we can isolate a larger class, the *plain partial weight assignments*, for which the  $(H, C, K)$ -problem problem is in class FPT.

One of the fundamental tools for designing FPT algorithms for decision problems is the so called *Reduction to problem kernel*. The method consists in an FPT self-reduction (in polynomial time in  $n$  and  $k$ ) that transforms any problem input  $x$  to a reduced instance  $f(x)$  of the same problem, the kernel, such that the size of  $f(x)$  depend *only* on some function of  $k$ . (See [6] for discussions on fixed parameter tractability and the ways of constructing kernels.)

In recent years, there has been a big effort focused on developing a theory of intractability for parameterized counting problems (see for example [10,

14]). However, so far, little progress has been done in the development of algorithmic techniques for counting problems. We introduce a new algorithmic tool denoted the *compactor enumeration* that, we expect, will play a role similar to kernelization for counting problems. We define a general type of structure, called *compactor*, which contains a *certificate* for each set in a suitable partition of the solution space. The size of such a structure (as well as the sizes of its elements) depends only on the parameterized part of the problem. Formally, given a parameterized problem  $\Pi$ , where  $k$  is the size of the parameter, let  $\text{Sol}(\Pi, k)$  be the set of solutions. We say that  $\text{Cmp}(\Pi, k)$  is a *compactor* if

- $|\text{Cmp}(\Pi, k)|$  is a function that depends only on  $k$ .
- $\text{Cmp}(\Pi, k)$  can be enumerated with an FPT algorithm.
- There is a surjective function  $\mathbf{m} : \text{Sol}(\Pi, k) \rightarrow \text{Cmp}(\Pi, k)$ .
- For any  $\mathfrak{C} \in \text{Cmp}(\Pi, k)$ ,  $|\{\mathbf{m}^{-1}(\mathfrak{C})\}|$  can be computed by an FPT algorithm.

Observe that if the above conditions hold, by *enumerating* all certifying structures and computing, for each of them, the number of solutions they certify, we obtain a FPT algorithm for counting the solutions of the initial problem.

We stress that our method is completely different than the method of reduction to a problem kernel as, in our approach, we essentially reduce the counting problem to an enumeration problem. Moreover, in the reduced problem the instance is not a subset of the instances of the initial problem as in the kernel reduction case. In some of our applications the compactor is a structure consisting of suitable collection of functions. We stress that our new method is also applicable to parameterized decision problems where the technique of reduction to a kernel seems to fail, like for instance the  $(H, C, K)$ -coloring (see Section 4).

The complexity of all our FPT algorithms for counting are *linear* in  $n$ , and it is of the form  $O(f_1(k)n + f_2(k)g + f_3(k)\log n + f_4(k))$ , where  $g$  is the number of connected components of  $G$  and functions  $f_i$  increase in complexity as  $i$  grows. Although the counting algorithms can be used to solve the decision version of the problem, we design specific algorithms for the decision version where the contribution of the parameter is similar of with lower order, with the exception that the  $\log n$  term disappears. To achieve this fast FPT algorithms for particular sub cases of our problems we combine compactors with different design tools like reduction to a kernel, *bounded search trees*, and *parameterized reduction* (see for example [15] for a survey on tools for parameterizing algorithms).

Finally, let us mention that several parameterized problems can be modeled as particular cases of the  $(H, C, K)$ -coloring, in which  $(H, C, K)$  is simple. Therefore, the techniques developed in this paper can be used to produce FPT algorithms for both the counting and the decision versions. For instance counting the number of vertex covers of size  $k$  is known to be in FPT by [10], the result can be easily obtained from this paper. Also, the following parameterization of PARTITION: “Given integers  $x_1, \dots, x_n$  and a parameter  $k$ , is there a set  $I \subseteq [n]$  such that  $\sum_{i \in I} x_i \geq k$  and  $\sum_{i \notin I} x_i \geq k$ ?” has an FPT algorithm for both counting and decision problems.

It is a remaining open question whether the classes of parameterized assignments that we present in this paper are the wider possible that correspond to parameterized problems solvable by FPT algorithms.

## 2 Definitions

We use standard notation for graphs. Following the terminology in [8,9] we say that a graph is *reflexive* if all its vertices have a loop, and that a graph is *irreflexive* when none of its vertices is looped. We denote by  $g = g(G)$  the number of connected components of  $G$ . For  $U \subseteq V(G)$ , we use  $G - U$  as a notation for the graph  $G[V(G) - U]$ . Given two graphs  $G$  and  $G'$ , define  $G \cup G' = (V(G) \cup V(G'), E(G) \cup E(G'))$ , and  $G \oplus G' = (V(G) \cup V(G'), E(G) \cup E(G') \cup \{\{u, v\} \mid u \in V(G), v \in V(G')\})$ . Given a bipartite connected graph  $G$ , the vertex set  $V(G)$  is splitted among two sets  $X(G)$  and  $Y(G)$  so that every edge has an end point in the  $X$ -part and another in the  $Y$ -part. Observe that the  $X$  and  $Y$  parts of a connected bipartite graph are defined without ambiguity.  $K_{r,s}$  denotes the complete irreflexive bipartite graph on two sets with  $r$  and  $s$  vertices each, notice that  $K_{r,0}$  denotes a complete bipartite graph, and  $K_n^r$  denotes the reflexive clique with  $n$  vertices.  $N_G(v)$  denotes the set of neighbors of  $v$  in the graph  $G$ .

Given two functions  $\varphi : A \rightarrow \mathbb{N}$  and  $\psi : A' \rightarrow \mathbb{N}$ , with  $A' \cap A = \emptyset$ , define  $(\varphi \cup \psi)(x)$  to be  $\varphi(x)$ , for  $x \in A$ , and  $\psi(x)$ , for  $x \in A'$ . Given two functions  $\varphi, \psi : A \rightarrow \mathbb{N}$ ,  $\varphi + \psi$  denotes its sum and we say that  $\phi \leq \psi$  if, for any  $x \in A$ ,  $\phi(x) \leq \psi(x)$ . We denote by  $\emptyset$  the empty function. We often need contiguous sets of indices, for  $n, m \in \mathbb{N}$ , we define  $[n] = \{1, \dots, n\}$ ,  $[-m] = \{-m, \dots, -1\}$ ,  $[-m, n] = \{-m, \dots, -1, 0, 1, \dots, n\}$ , and  $\langle n \rangle = \{0, 1, \dots, n\}$ .

Given two graphs  $G$  and  $H$  we say that  $\chi : V(G) \rightarrow V(H)$  is a  $H$ -coloring of  $G$  if for any  $\{x, y\} \in E(G)$ ,  $\{\chi(x), \chi(y)\} \in E(H)$ .

A *partial weight assignment* on a graph  $H$  is a triple  $(H, C, K)$ , where  $C \subseteq V(H)$  and  $K : C \rightarrow \mathbb{N}$ . We refer to the vertices of  $C$  as the *weighted vertices*. For each partial weight assignment  $(H, C, K)$ , we use the notations  $S = V(H) - C$ ,  $h = |V(H)|$ ,  $c = |C|$ ,  $s = |S|$  and  $k = \sum_{c \in C} K(c)$ . We say that a  $(H, C, K)$  is *positive* if  $K(a) \neq 0$  for any  $a \in C$ .

Given a partial weight assignment  $(H, C, K)$ ,  $\chi : V(G) \rightarrow V(H)$  is a  $(H, C, K)$ -coloring of  $G$  if  $\chi$  is an  $H$ -coloring of  $G$  such that for all  $a \in C$ , we have  $|\chi^{-1}(a)| = K(a)$ . We denote by  $\mathcal{H}_{(H, C, K)}(G)$  the set of all the  $(H, C, K)$ -colorings of  $G$ , often we drop the subscript when  $(H, C, K)$  is clear from the context. For a fixed partial weight assignment  $(H, C, K)$ , the  $(H, C, K)$ -coloring problem asks whether there exists an  $(H, C, K)$ -coloring of an input graph  $G$ . In this definition the parameters are the integers in the images of  $K$ . The parameterized independent set and vertex cover problems are particular instances of the  $(H, C, K)$ -coloring problem.

For a fixed graph  $H$ , given a graph  $G$ , a  $(H, G)$ -list is a function  $L : V(G) \rightarrow 2^{V(H)}$  mapping any vertex of  $G$  to a subset of  $V(H)$ . For any  $v \in V(G)$ , the *list* of  $v$  is the set  $L(v) \subseteq V(H)$ . The *list  $H$ -coloring* problem asks whether, given a graph  $G$  and an  $(H, G)$ -list  $L$ , there is an  $H$ -coloring  $\chi$  of  $G$  so that

for every  $u \in V(G)$ ,  $\chi(u) \in L(u)$ . This problem can be parameterized in the same way as the  $H$ -coloring to define the *list*  $(H, C, K)$ -coloring and the *list*  $(H, C, \leq K)$ -coloring problems. We denote by  $\mathcal{H}_{(H, C, K)}(G, L)$  the set of all the  $(H, C, K)$ -colorings of  $(G, L)$ . As before we will use  $\mathcal{H}(G, L)$  when  $(H, C, K)$  is clear from the context.

As usual, the  $\#(H, C, K)$ -coloring and the list  $\#(H, C, K)$ -coloring, problems denote the counting versions of the corresponding decision problems.

Notice that in contraposition to the  $H$ -coloring, a  $(H, C, K)$ -coloring, in general, cannot be obtained as the disjoint union of  $(H, C, K)$ -colorings of the connected components of an input graph  $G$ . This is expressed in the next lemmas, whose proof is left to the reader.

**Lemma 1.** *Let  $G$  be a graph with connected components  $G_i$ ,  $i = 1, \dots, g$ , and let  $\chi_i$  be a  $(H, C, K_i)$ -coloring of  $G_i$ , for any  $i$ ,  $1 \leq i \leq g$ . Then,  $\chi^* = \chi_1 \cup \dots \cup \chi_t$  is a  $(H, C, K_1 + \dots + K_t)$ -coloring of  $G$ .*

**Lemma 2.** *Let  $(H, C, K)$  and  $(H', C', K')$  be two partial weight assignments. Let  $G$  and  $G'$  be two graphs, let  $\chi$  be an  $(H, C, K)$ -coloring of  $G$  and let  $\chi'$  be an  $(H', C', K')$ -coloring of  $G'$ . Then  $\chi \cup \chi'$  is a  $(H \cup H', C \cup C', K \cup K')$ -coloring of  $G \cup G'$ .*

Finally, we define some types of partial weight assignments on connected graphs. Let  $H$  be a connected graph, a partial weight assignment is a *1-component* if  $E(H - C) = \emptyset$ ; a *2-component* if  $H$  is a reflexive clique; a *3-component* if  $H$  is a complete bipartite graph and  $C \subsetneq Y(G)$ ; a *4-component* if  $H[C]$  is a reflexive clique with all its vertices adjacent with one looped vertex of  $H - C$ ; and a *6-component* if  $H$  is bipartite,  $C \subsetneq Y(H)$ , there is a vertex  $x \in X(H)$  adjacent to all the vertices in  $C$ , and at least one vertex in  $Y(H) - C$ .

A partial weight assignment  $(H, C, K)$  is said to be *simple* when each connected component  $H'$  is either a 1-component, a 2-component or a 3-component; it is said to be *plain* when each connected component  $H'$  of  $H$  is either a 1-component, a 4-component, or a 6-component. Observe that the class of simple partial weight assignments is a generalization of the class of *compact* partial weight assignments used in [5].

### 3 Counting $(H, C, K)$ -Colorings of Connected Graphs

We introduce some vertex-weighted graphs and extend the notion of list  $(H, C, K)$ -coloring to them. A *tribal graph* is a graph  $G$  together with a positive weight assignment  $p : V(G) \rightarrow \mathbb{N}^+$ . The vertices in the set  $T(G) = \{v \in V(G) \mid p(v) > 1\}$  are called *tribes* of  $G$  and the vertices in  $I(G) = V(G) - T(G)$  *individuals* of  $G$ . The *expansion*  $\widehat{G}$  of a tribal graph is a graph in which each  $v \in T(G)$  is replaced by  $p(v)$  copies of  $v$ . We associate a tribal graph with a subset of vertices of a graph, typically with “few” vertices, and an equivalence relation on the remaining vertices, represented by tribes. Our compactors are obtained by restricting the maximum weight of a tribe, in the following way:



For a non negative integer  $r$ , the  $r$ -restriction of a tribal graph  $(G, p)$ , is the tribal graph  $(G, p')$  where, for any  $v \in V(G)$ ,  $p'(v) = \min\{p(v), r\}$ . For sake of readability we denote a tribal graph as  $\tilde{G}$ , eliminating the explicit reference to the node weights.

A list  $(H, C, K)$ -coloring of a tribal graph  $\tilde{G}$  and an  $(H, \tilde{G})$ -list  $L$  is a mapping  $w : V(\tilde{G}) \times V(H) \rightarrow \langle k \rangle$  where: (1) for any  $v \in V(\tilde{G})$  and  $a \in S$ ,  $w(v, a) \leq 1$ ; (2) for any  $v \in V(\tilde{G})$  and  $a \in C$ ,  $w(v, a) \leq K(a)$ ; (3) for any  $v \in V(\tilde{G})$ ,  $1 \leq \sum_{a \in H} w(v, a) \leq p(v)$ ; (4) if  $\{v, u\} \in E(\tilde{G})$  then for all  $a, b \in H$  with  $w(v, a) > 0$  and  $w(u, b) > 0$ ,  $\{a, b\} \in E(H)$ ; (5) for any  $a \in C$ ,  $\sum_{v \in V(\tilde{G})} w(v, a) = K(a)$ ; and (6) for any  $v \in V(\tilde{G})$  and  $a \in V(H)$  with  $w(v, a) > 0$ ,  $a \in L(v)$ .

Notice that in the case  $C = \emptyset$  the above definition coincides with the concept of list  $H$ -coloring. Moreover, in case that  $p(v) = 1$  for any  $v \in V(\tilde{G})$ , we obtain a list  $(H, C, K)$ -coloring of  $(\tilde{G}, L)$  by assigning to each vertex in  $\tilde{G}$  the unique vertex in  $V(H)$  for which  $w(v, a) = 1$ . Also, by removing condition 6 we get a definition of  $(H, C, K)$ -coloring of a tribal graph.

The following result gives some initial results on colorings for tribal graphs. We use  $\tilde{n}$  ( $\tilde{m}$ ) to denote the number of vertices (edges) of a tribal graph  $\tilde{G}$ .

**Lemma 3.** *Let  $(H, C, K)$  be a partial weight assignment. Let  $\tilde{G}$  be a tribal graph and let  $L$  be a  $(H, \tilde{G})$ -list. Given  $w : V(\tilde{G}) \times V(H) \rightarrow \langle k \rangle$  we can check whether  $w$  is a list  $(H, C, K)$ -coloring of  $(\tilde{G}, L)$  in time  $O(h^2 \tilde{n} + \tilde{m})$ . Furthermore,  $|\{w : V(\tilde{G}) \times V(H) \rightarrow \langle k \rangle\}| \leq ((k+1)^c 2^s)^{\tilde{n}}$ .*

Our next step is to introduce an equivalence relation on the vertices of  $G$  and construct a tribal graph from there. Let  $(H, C, K)$  be a partial weight assignment, for a given graph  $G$  together with a  $(H, G)$ -list  $L$ , define  $\mathcal{P}$  to be the partition of  $V(G)$  induced by the equivalence relation,

$$v \sim u \text{ iff } [N_G(v) = N_G(u) \wedge L(v) = L(u)].$$

For  $v \in V(G)$ ,  $P_v$  denotes the set  $\{u \mid u \sim v\}$  and for any  $Q \in \mathcal{P}$ , we select a representative vertex  $v_Q \in Q$ .

In general, we keep all the vertices in some classes as individuals and one representative of each remaining classes as tribes. We say that  $R \subseteq V(G)$  is a *closed set* if for any  $v \in R$  we have  $P_v \subseteq R$ . Let  $R$  be a closed set, the *tribal graph associated to  $(G, R)$*  is  $\tilde{G} = (G[R \cup \{v_Q \mid Q \cap R = \emptyset\}], p)$ , where  $p(v) = 1$  when  $v \in R$ , and  $p(v_Q) = |Q|$  for any  $Q \cap R = \emptyset$ .

Let  $\tilde{G}_{k+s}$  be the  $k+s$ -restriction of the tribal graph  $\tilde{G}$  associated to  $G$  and some closed set  $R$ . The following lemmatta show the properties that will allow us to use  $\mathcal{H}(\tilde{G}_{k+s}, L)$  as a compactor for  $\mathcal{H}(G, L)$  in several cases. Observe that as  $V(\tilde{G})$  can be seen as a subset of  $V(G)$ , so we can keep the name of the associated  $(H, G)$  list as  $L$  in order to simplify notation.

**Lemma 4.** *Let  $(H, C, K)$  be a partial weight assignment. Given a graph  $G$  together with a  $(H, G)$ -list  $L$  and a closed set  $R$ , let  $\tilde{G}$  be the tribal graph associated to  $(G, R)$ . Then, there is a surjective function from  $\mathcal{H}(G, L)$  into  $\mathcal{H}(\tilde{G}_{k+s}, L)$ .*

Given a list  $(H, C, K)$ -coloring  $w$  of  $(\tilde{G}_{k+s}, L)$ , let  $NE(w)$  be  $|\{\chi \in \mathcal{H}(G, L) \mid w = w_\chi\}|$ , for any  $v \in T(\tilde{G})$  define the function  $W$

$$W(w, v) = \binom{|P_v|}{\tau} \frac{\tau!}{\prod_{a \in C} w(v, a)!} \binom{|P_v| - \tau}{\sigma} \sigma! \sigma^{|P_v| - \tau - \sigma},$$

where  $\sigma = \sum_{a \in S} w(v, a)$  and  $\tau = \sum_{a \in C} w(v, a)$ . In the next result we show how to compute  $NE(w)$  using the function  $W$ . As it is usual when dealing with a counting problem, we assume that all basic arithmetic operations take constant time.

**Lemma 5.** *Let  $(H, C, K)$  be a partial weight assignment. Given a graph  $G$  together with a  $(H, G)$ -list  $L$  and a closed set  $R$ , let  $\tilde{G}$  be the tribal graph associated to  $(G, R)$  and let  $w$  be a list  $(H, C, K)$ -coloring of  $(\tilde{G}_{k+s}, L)$ . Then,  $NE(w) = \prod_{v \in T(\tilde{G})} W(w, v)$ . Furthermore, for any  $v \in T(\tilde{G})$ , given  $w$ ,  $\tilde{G}$  and  $|P_v|$ , the value  $W(w, v)$  can be computed in  $O(\log h + c \log k + \log n)$ .*

Now we have shown most of the required compactor properties, it remains only to show that in the particular classes of partial weight assignments the obtained compactor is small. Before going into that we present the central idea the algorithm **Count-List-Colorings** that we will adapt later to the different cases. The idea of the algorithm is given  $w \in \mathcal{H}(\tilde{G}_{k+s}, L)$  we have to compute the number of possible extensions to a list  $(H, C, K)$ -coloring of  $(G, L)$  and, according to Lemmata 4 and 5, the overall sum of these numbers will be the number of the list  $(H, C, K)$ -colorings of  $(G, L)$ .

**function** Count-List-Colorings( $H, C, K, G, L, R$ )

**input** A partial weight assignment  $(H, C, K)$  on  $H$  such that  $E(H - C) = \emptyset$

A graph  $G$  and a  $(H, G)$ -list  $L$ ,

**output** The number of  $(H, C, K)$ -colorings of  $(G, L)$ .

**begin**

$GT = \text{Tribal}(H, C, K, G, L, R)$

$GTR = \text{Restrict}(GT, k + s)$

$\nu = 0$

**for all**  $w : V(GTR) \times V(H) \rightarrow \{0, \dots, k\}$  **do**

**if**  $w \in \mathcal{H}(GTR, L)$

**then**

$\nu = \nu + \prod_{v \in T(GT)} W(w, v)$

**end if**

**end for**

**return**  $\nu$

**end**

**Theorem 6.** *Let  $(H, C, K)$  be a partial weight assignment. Given a graph  $G$  together with a  $(H, G)$ -list  $L$  and a closed set  $R$ . Then, algorithm **Count-List-Colorings** outputs the number of list  $(H, C, K)$ -colorings of  $(G, L)$  within  $O(n(n +$*

$h) + (n+h)2^{n+h} + (h^2\tilde{n} + \tilde{m})((k+1)^c 2^s)^{\tilde{n}}$ , steps, where  $\tilde{n}$  and  $(\tilde{m})$  are the number of vertices (edges) of the tribal graph  $\tilde{G}$  associated to  $(G, R)$ .

In the following we specify the particular definition of the equivalence relation for the case of 1-components and describe the main result for 2 and 3-components.

In the case of 1-components the compactor is obtained after splitting the vertices of  $G$  according to degree and connectivity properties. Given a graph  $G$  and an integer  $k$ , the  $k$ -splitting of  $G$  is a partition of  $V(G)$  in three sets,  $(R_1, R_2, R_3)$  where  $R_1$  is the set of vertices in  $G$  of degree  $> k$ ,  $R_2$  is formed by the non isolated vertices in  $G' = G[V(G) - R_1]$  and  $R_3$  contains the isolated vertices in  $G'$ .

**Lemma 7.** *Let  $(H, C, K)$  be a partial weight assignment, where  $H - C$  is edge-less. Given a graph  $G$ , let  $(R_1, R_2, R_3)$  be the  $k$ -splitting of  $G$ . If  $|R_1| > k$  or  $|R_2| > k^2 + k$ , then  $\mathcal{H}(G, L) = \emptyset$ . Furthermore, the  $k$ -splitting of  $G$  can be computed in  $O(kn)$  steps.*

The next result is obtained taking into account that all the vertices in  $R_3$  have degree at most  $k$  and that for any  $v \in R_3$  we have  $N_G(v) \subseteq R_1$ .

**Lemma 8.** *Let  $(H, C, K)$  be a partial weight assignment, where  $H - C$  is edge-less. Given a graph  $G$  and a  $(H, G)$ -list, let  $(R_1, R_2, R_3)$  be the  $k$ -splitting of  $G$ . Then,  $R = R_1 \cup R_2$  is a closed set. Furthermore, the tribal graph  $\tilde{G}$  associated to  $(G, R)$  has at most  $k^2 + 2k + 2^{k+h}$  vertices and  $(k^2 + 2k)^2 + k2^{k+h}$  edges, and  $\tilde{G}$  can be computed in  $O((h+k)n + (h+k)2^{k+h})$  steps.*

Using the previous lemmatta we can compute  $|\mathcal{H}(G, L)|$  in FPT.

**Theorem 9.** *Let  $(H, C, K)$  be a partial weight assignment, where  $H - C$  is edge-less. Given a graph  $G$  and a  $(H, G)$ -list  $L$ , let  $(R_1, R_2, R_3)$  be the  $k$ -splitting of  $G$  and let  $\tilde{G}$  be the tribal graph associated to  $(G, R_1 \cup R_2)$ . Then,  $|\mathcal{H}(G, L)| = \sum_{w \in \mathcal{H}(\tilde{G}, L)} NE(w)$ . Furthermore,  $|\mathcal{H}(G, L)|$  can be computed within  $O((h+k)n + (h^2k^2 + k^4 + (\log h + c \log k + \log n)2^{h+k})((k+1)^c 2^s)^{k^2+2k+2^{h+k}})$  steps.*

In the case of 2-components we use the following splitting: Given a graph  $G$  together with a  $(H, G)$ -list  $L$ , the list splitting of  $(G, L)$  is a partition of  $V(G)$  in two sets,  $(N_1, N_2)$  where  $N_1 = \{v \in V(G) \mid L(v) \cap S = \emptyset\}$ . Let  $G' = (V(G), \emptyset)$ .

**Theorem 10.** *Let  $(H, C, K)$  be a partial weight assignment with  $H = K_h^r$ . Given an edge-less graph  $G$  and a  $(H, G)$ -list  $L$ , let  $(N_1, N_2)$  be the list-splitting of  $(G, L)$ . Then, if  $|N_1| > k$ , there are no  $(H, C, K)$ -colorings of  $(G, L)$ . Otherwise,  $N_1$  is a closed set and the tribal graph  $\tilde{G}$  associated to  $(G', N_1)$  has at most  $k + 2^h$  vertices and no edges, and  $\mathcal{H}(G, L) = \mathcal{H}(G', L)$ . Furthermore,  $|\mathcal{H}(G, L)|$  can be computed within  $O(hn + 2^h + (h^2(k + 2^h) + (\log h + c \log k + \log n)2^h)((k+1)^c 2^s)^{k+2^h})$  steps.*

In the case of 3-components we use the following splitting: Let  $(H, C, K)$  be a partial weight assignment with  $H = K_{xy}$  and  $C \subseteq Y(H)$ . Given a bipartite graph  $G$  together with a  $(H, G)$ -list  $L$ . The *bipartite splitting* of  $(G, L)$  with respect to  $(H, C, K)$  is a partition of  $V(G)$  in three sets,  $(M_1, M_2, M_3)$  where  $M_3 = \{v \in V(G) \mid L(v) \cap S \neq \emptyset\}$ ,  $M_1 = X(G) - M_3$  and  $M_2 = Y(G) - M_3$ . Given a connected bipartite graph  $G$ , let  $G' = (X(G), Y(G), X(G) \times Y(G))$ .

**Theorem 11.** *Let  $(H, C, K)$  be a partial weight assignment with  $H = K_{xy}$  and  $C \subseteq Y(H)$ . Given a complete bipartite graph  $G$  together with a  $(H, G)$ -list  $L$ , let  $(M_1, M_2, M_3)$  be the bipartite list partition of  $(G, L)$ . Then, if  $|M_1| > 0$  and  $|M_2| > 0$ , or  $|M_1| + |M_2| > k$ , there are no list  $(H, C, K)$ -colorings of  $(G, L)$ . Otherwise,  $M = M_1 \cup M_2$  is a closed set, the tribal graph  $\tilde{G}$  associated to  $(G', M)$  has at most  $k + 4^h$  vertices and  $(k + 2^h)2^h$  edges, it can be computed in  $O(hn + (h + 1)2^h)$  steps, and  $\mathcal{H}(G, L) = \mathcal{H}(G', L)$ . Furthermore,  $|\mathcal{H}(G, L)|$  can be computed within  $O(hn + (h + 1)2^h + (h^2(k + 4^h) + (k + 2^h)2^h + (\log h + c \log k + \log n)4^h) ((k + 1)c2^s)^{k+4^h})$  steps.*

## 4 Counting List $(H, C, K)$ -Coloring for Simple Weight Assignments

In this section we provide an FPT algorithm for counting the number of list  $(H, C, K)$ -coloring when all the connected components of  $H$  are 1, 2 or 3-components. The graph  $H$  is decomposed as the disjoint union of  $\gamma$  components,  $H_{-\rho}, \dots, H_\eta$ , where  $H_0$  is the disjoint union of the 1-components, for  $\iota \in [\eta]$ ,  $H_\iota$  is a 2-component, while for  $\iota \in [-\rho]$ ,  $H_\iota$  is a 3-component.

Furthermore, the graph  $G$  may have different connected components, at least one for each component of  $H$  that contains a vertex in  $C$ . We assume that  $G$  has  $g$  connected components,  $\{G_1, \dots, G_g\}$ . Given a  $(H, G)$ -list  $L$ , we often have to speak of a list coloring of a component  $G_i$ , we will refer to this as a  $(G_i, L)$  coloring, understanding that we keep the original list restricted to the adequate sets. Furthermore, we assume that  $G_i$  is always mapped to a connected component of  $H$ .

We need further definitions to establish the form and properties of the compactor. For any  $i \in [g]$  and  $\iota \in [-\rho, \eta]$ ,  $\tilde{G}_{i\iota}$  denotes the tribal graph obtained from  $(G_i, L_{i\iota})$  through the splitting  $(k, \text{list or bipartite})$  corresponding to the type of  $(H_\iota, C_\iota, K_\iota)$ . The vertices in  $\tilde{G}_{i\iota}$  are relabeled so that if  $n_{i\iota} = |V(\tilde{G}_{i\iota})|$ , then  $V(\tilde{G}_{i\iota}) = [n_{i\iota}]$  and, for any  $i$ ,  $1 \leq i < n_{i\iota}$ ,  $p_{i\iota}(i) \leq p_{i\iota}(i + 1)$ .

For any  $i \in [g]$ ,  $\iota \in [-\rho, \eta]$ , and  $w : [n_{i\iota}] \times V(H_\iota) \rightarrow [k_\iota]$  set  $\beta(i, \iota, w) = 1$  or 0 depending on whether  $w$  is a list  $H_\iota$ -coloring of  $\tilde{G}_{i\iota}$  or not.

For any  $\iota \in [-\rho, \eta]$  we define the following equivalence relation on  $[g]$ :

$$\begin{aligned} i \sim_\iota j \text{ iff } n_{i\iota} = n_{j\iota} \text{ and } \forall v \in [n_{i\iota}] \ p_{i\iota}(v) = p_{j\iota}(v) \\ \text{and } \forall w : [n_{i\iota}] \times V(H_\iota) \rightarrow [k_\iota] \ \beta(i, \iota, w) = \beta(j, \iota, w). \end{aligned}$$

We say that two components of  $G$  are equivalent when they are equivalents with respect to all the components of  $H$ . Formally, for  $i, j \in [g]$ ,  $i \sim j$  iff  $\forall \iota \in [-\rho, \eta]$   $i \sim_\iota j$ . For  $i \in [g]$ , let  $Q_i$  denote the equivalence class of  $i$ .

Let  $\mathcal{Q}$  be the partition induced in  $[g]$  by  $\sim$ . Notice that we have at most

$$\left( (k+s)(k^2 + 2k + 2^{k+h}) k^{(k+s)(k^2+2k+2^{k+h})} 2^{h^{(k+s)(k^2+2k+2^{k+h})}} \right)^\gamma$$

equivalence classes. Each class can be represented by a sequence of  $\gamma$  triples  $(n_\iota^Q, p_\iota^Q, \beta_\iota^Q)$ , formed by an integer, a tribal weight, and a binary string of length at most  $h^{(k+s)(k^2+2k+2^{k+h})}$ , representing the size of the vertex set, the size of the tribes, and the associated mask respectively.

For any  $Q \in \mathcal{Q}$  define

$$\begin{aligned} \mathcal{F}_0(Q) &= \cup_{\iota \in [-\rho, \eta]} \{w : [n_\iota^Q] \times V(H_\iota) \rightarrow [k] \mid \beta_\iota^Q(w) = 1 \text{ and } w^{-1}(C) = \emptyset\}, \text{ and} \\ \mathcal{F}_1(Q) &= \cup_{\iota \in [-\rho, \eta]} \{w : [n_\iota^Q] \times V(H_\iota) \rightarrow [k] \mid \beta_\iota^Q(w) = 1 \text{ and } w^{-1}(C) \neq \emptyset\} \end{aligned}$$

The compactor is defined as

$$\begin{aligned} \mathcal{F}(G, L) &= \{((A_Q, o_Q, B_Q))_{Q \in \mathcal{Q}} \mid \\ &\quad \forall Q \in \mathcal{Q} \ A_Q \subseteq \mathcal{F}_1(Q), o_Q : A_Q \rightarrow [k], B_Q \subseteq \mathcal{F}_0(Q) \\ &\quad \forall a \in C \ \sum_{Q \in \mathcal{Q}} \sum_{w \in A_Q} o(w) |w^{-1}(a)| = K(a)\} \end{aligned}$$

**Lemma 12.** *Let  $(H, C, K)$  be a simple partial weight assignment. Given a graph  $G$  together with a  $(H, G)$ -list  $L$ . Then, there is a surjective mapping from  $\mathcal{H}(G, L)$  into  $\mathcal{F}(G, L)$ .*

It remains to show that the number of preimages, in the above function, of a sequence  $\sigma \in \mathcal{F}(G, L)$  can be computed in FPT. To do so, it is enough to guarantee that, for each equivalence class  $Q$ , the number of ways to assign the colorings in  $(A_Q, o_Q, B_Q)$  to all the members of  $Q$  can be computed in FPT. We show this fact using a dynamic programming approach. Once this is done all the properties of the compactor are fulfilled and we get

**Theorem 13.** *Let  $(H, C, K)$  be a simple partial weight assignment. Given a graph  $G$  together with a  $(H, G)$ -list  $L$ , then  $|\mathcal{H}(G, L)|$  can be computed in time  $O(\gamma(k+h)n + f_1(k, h)g + f_2(k, h)\log n + f_3(k, h))$ , for suitable functions.*

## 5 Decision Problems

Let us consider the decision version of the list  $(H, C, K)$ -coloring problem for simple partial weight assignments: we can use a weak form of compactorization to obtain a faster FPT algorithm. Due to lack of space we only comment on the main trends. For the case of connected graphs we compute the  $k+1$ -restriction of the

corresponding tribal graph  $\tilde{G}$  (depending on the type of component) according to the definitions given in Section 3, and show that there is list  $(H, C, K)$ -coloring of  $(G, L)$  if and only if there is a list  $(H, C, K)$ -coloring of  $(\tilde{G}_{k+1}, L)$ . The complexity of the algorithms is summarized in the following result.

**Lemma 14.** *Let  $(H, C, K)$  be a partial weight assignment, let  $G$  be a graph, and let  $L$  be a  $(H, G)$ -list. When  $G$  has no edges or  $H = K_h^r$ , the existence of a list  $H$ -coloring of  $G$  can be decided in  $O(hn)$  steps, while the existence of a list  $(H, C, K)$ -coloring of  $(G, L)$  can be decided in  $O((k+h)n + nk \log k \sqrt{n+k})$  steps. When  $H - C$  is edge-less, the existence of a list  $(H, C, K)$ -coloring of  $(G, L)$  can be decided in  $O(2^k c^k [(k+h)n + nk \log k \sqrt{n+k}])$  steps.*

When  $G$  is not connected, we make use of a more involved compactor. However, as before, we only need to check that the compactor is not empty to check the existence of a list  $(H, C, K)$ -coloring of  $(G, L)$ . To define the compactor we use partial weight assignments for any component of  $H$ , so for any  $\iota \in [-\rho, \eta]$ , define  $\mathcal{W}_\iota = \{W : C \rightarrow \mathbb{N} \mid \mathbf{0} \leq W \leq K_\iota\}$ . Let  $k_\iota^* = \prod_{a \in C_\iota} K_\iota(a)$ . Notice that  $|\mathcal{W}_\iota| = k_\iota^*$  and that  $k^* = \prod_{a \in C} K(a) = \prod_{\iota \in [-\rho, \eta]} k_\iota^*$ . Finally, let  $\mathcal{W}_\iota^+ = \mathcal{W}_\iota - \{\mathbf{0}\}$ .

Our next definition will be the basis for classifying the components of  $G$  according to the combinations of components of  $H$  and partial weight assignments with correct list colorings. Given  $i \in [g]$ ,  $\iota \in [-\rho, \eta]$ , and  $W \in \mathcal{W}_\iota$  define  $\alpha(i, \iota, W) = 1$  or 0 depending on whether there is a list- $(H_\iota, C_\iota, W)$ -coloring of  $(\tilde{G}_{i\iota}, L_{i\iota})$  or not.

We isolate the information that has to capture a set of components of  $G$  to certify the existence of a list coloring. Given  $A \subseteq [g]$ , and  $\iota \in [-\rho, \eta]$  define

$$\mathcal{F}(A, \iota) = \{(Z, f) \mid Z \subseteq A, |Z| \leq k_\iota, f : Z \rightarrow \mathcal{W}_\iota^+, \sum_{i \in Z} f(i) = K_\iota \wedge \forall i \in Z \alpha(i, \iota, f(i)) = 1\}, \text{ and the compactor is}$$

$$\mathcal{F}(A) = \{(Z_{-\rho}, f_{-\rho}), \dots, (Z_\eta, f_\eta) \mid Z_1 \dots Z_\eta \text{ form a non trivial partition of } A, \forall \iota \in [-\rho, \eta] f_\iota : Z_\iota \rightarrow \mathcal{W}_\iota \wedge (Z_\iota, f_\iota) \in \mathcal{F}(A, \iota)\}.$$

Thus we can establish the main property of the compactor  $\mathcal{F}(A)$ . We say that  $A \subseteq [g]$  is *proper* if  $\mathcal{F}(A) \neq \emptyset$  and for any  $i \in [g] - A$ , there is  $\iota \in [-\rho, \eta]$  such that  $\alpha(i, \iota, \mathbf{0}) = 1$ .

We establish an equivalence relation to reduce the number of candidate sets as follows

$$i \sim j \text{ iff } \forall \iota \in [-\rho, \eta] \forall W \in \mathcal{W}_\iota \alpha(i, \iota, W) = \alpha(j, \iota, W).$$

Let  $\mathcal{P}$  be the partition induced by the equivalence classes. Notice that we have at most  $2^{k^*}$  equivalence classes and that each class can be represented by a binary string of length  $k^*$ . Let  $M \subseteq [g]$  be obtained by keeping  $k+1$  representatives from each class with more than  $k$  members, all the class otherwise.

**Lemma 15.** *Let  $(H, C, K)$  be a simple partial weight assignment. Given a graph  $G$  together with a  $(H, G)$ -list  $L$ . Then,  $\mathcal{H}_{(H, C, K)}(G, L) \neq \emptyset$  if and only if there is a  $A \subseteq M$  that is proper.*

Finally, by generating all the sets  $A \subseteq M$  and all the elements in  $\mathcal{F}(A)$  and performing the adequate test, for suitable functions  $f'_2$  and  $f'_4$  of lower order than  $f_2$  and  $f_4$  that appear in Theorem 13 respectively, we have

**Theorem 16.** *For simple partial weight assignment  $(H, C, K)$ , the list  $(H, C, K)$ -coloring problem can be solved in time  $O(\gamma(k+h)n + f'_2(k, h)g + f'_4(k, h))$ .*

Our results can be extended to provide a FPT algorithm for the  $(H, C, K)$ -coloring problem in the case of plain partial weight assignments. This goal is achieved through a series of parameterized reductions, from the general form of components to the *small* or *tiny* forms that are simple. We only need to extend to the case of 3-components the reductions presented in [2].

**Theorem 17.** *For any plain partial weight assignment  $(H, C, K)$ , the  $(H, C, K)$ -coloring problem can be solved in  $O(kn + f''_4(k, h))$  steps.*

## References

1. J. Díaz.  $H$ -colorings of graphs. *Bulletin of the European Association for Theoretical Computer Science*, (75):82–92, 2001.
2. J. Díaz, M. Serna, and D. Thilikos.  $(H, C, K)$ -coloring: Fast, easy and hard cases. *Mathematical Foundations of Computer Science*, vol. 2136 of LNCS, pages 304–315, 2001.
3. J. Díaz, M. Serna, and D. Thilikos. The restrictive  $H$ -coloring problem. *Discrete Applied Mathematics*, to appear.
4. J. Díaz, M. Serna, and D. M. Thilikos. Recent results on parameterized  $H$ -colorings *Graphs, Morphisms and Statistical Physics*, DIMACS series in Discrete Mathematics and Theoretical Computer Science, vol. 63 pp 65–85, 2004.
5. J. Díaz, M. Serna, and D. M. Thilikos. Complexity issues on Bounded Restrictive  $H$ -colorings. Submitted, 2004.
6. R. G. Downey and M. R. Fellows. *Parameterized complexity*. Springer-Verlag, New York, 1999.
7. M. Dyer and C. Greenhill. The complexity of counting graph homomorphisms. *Random Structures Algorithms*, 17:260–289, 2000.
8. T. Feder and P. Hell. List homomorphisms to reflexive graphs. *Journal of Combinatorial Theory (series B)*, 72(2):236–250, 1998.
9. T. Feder, P. Hell, and J. Huang. List homomorphisms and circular arc graphs. *Combinatorica*, 19:487–505, 1999.
10. J. Flum and M. Grohe. The parameterized complexity of counting problems. *SIAM Journal on Computing*, 33(4):892–922, 2004.
11. H. Gabow and R. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18:1013–1036, 1989.
12. P. Hell and J. Nešetřil. On the complexity of  $H$ -coloring. *Journal of Combinatorial Theory (series B)*, 48:92–110, 1990.
13. P. Hell and J. Nešetřil. Counting list homomorphisms and graphs with bounded degrees. *Graphs, Morphisms and Statistical Physics*, DIMACS series in Discrete Mathematics and Theoretical Computer Science, vol. 63, pp 105–112, 2004.
14. C. McCartin. Parameterized counting problems. *Mathematical Foundations of Computer Science*, vol. 2420 of LNCS, pages 556–567, 2002.
15. C. McCartin. *Contributions to Parameterized Complexity*. PhD thesis, Victoria University of Wellington, 2003.



# On Variable-Sized Multidimensional Packing

Leah Epstein<sup>1,\*</sup> and Rob van Stee<sup>2,\*\*</sup>

<sup>1</sup> School of Computer Science, The Interdisciplinary Center, Herzliya, Israel. lea@idc.ac.il.

<sup>2</sup> Centre for Mathematics and Computer Science (CWI), Amsterdam, The Netherlands.  
Rob.van.Stee@cwi.nl.

**Abstract.** The main contribution of this paper is an optimal bounded space online algorithm for variable-sized multidimensional packing. In this problem, hyperboxes must be packed in  $d$ -dimensional bins of various sizes, and the goal is to minimize the total volume of the used bins. We show that the method used can also be extended to deal with the problem of resource augmented multidimensional packing, where the online algorithm has larger bins than the offline algorithm that it is compared to. Finally, we give new lower bounds for unbounded space multidimensional bin packing of hypercubes.

## 1 Introduction

In this paper, we consider the problem of online variable-sized multidimensional bin packing. In the  $d$ -dimensional box packing problem, we receive a sequence  $\sigma$  of items  $p_1, p_2, \dots, p_n$ . Each item  $p$  has a fixed size, which is  $s_1(p) \times \dots \times s_d(p)$ . I.e.  $s_i(p)$  is the size of  $p$  in the  $i$ th dimension. We have an infinite number of bins, each of which is a  $d$ -dimensional unit hyper-cube. Each item must be assigned to a bin and a position  $(x_1(p), \dots, x_d(p))$ , where  $0 \leq x_i(p)$  and  $x_i(p) + s_i(p) \leq 1$  for  $1 \leq i \leq d$ . Further, the positions must be assigned in such a way that no two items in the same bin overlap. A bin is *empty* if no item is assigned to it, otherwise it is *used*. The goal is to minimize the number of bins used. Note that for  $d = 1$ , the box packing problem reduces to exactly the classic bin packing problem.

We focus on the *variable-sized* box packing problem, where bins of various sizes can be used to pack the items, and the goal is to minimize the total volume of all the bins used. To avoid a cumbersome notation, we assume that the available bins are hypercubes, although our algorithm can also be extended to the case where the bins are hyperboxes. The available sizes of bins (edge lengths) are denoted by  $\alpha_1 < \alpha_2 < \dots < \alpha_m = 1$ .

Items arrive *online*, which means that each item must be assigned in turn, without knowledge of the next items. We consider *bounded space* algorithms, which have the property that they only have a constant number of bins available to accept items at any point during processing. The bounded space assumption is a quite natural one, especially so in online box packing. Essentially the bounded space restriction guarantees that output of packed bins is steady, and that the packer does not accumulate an enormous backlog of bins which are only output at the end of processing.

---

\* Research supported by Israel Science Foundation (grant no. 250/01).

\*\* Research supported by the Netherlands Organization for Scientific Research (NWO), project number SION 612-30-002.



The standard measure of algorithm quality for box packing is the *asymptotic performance ratio*, which we now define. For a given input sequence  $\sigma$ , let  $\text{cost}_{\mathcal{A}}(\sigma)$  be the number of bins used by algorithm  $\mathcal{A}$  on  $\sigma$ . Let  $\text{cost}(\sigma)$  be the minimum possible number of bins used to pack items in  $\sigma$ . The *asymptotic performance ratio* for an algorithm  $\mathcal{A}$  is defined to be

$$\mathcal{R}_{\mathcal{A}}^{\infty} = \limsup_{n \rightarrow \infty} \sup_{\sigma} \left\{ \frac{\text{cost}_{\mathcal{A}}(\sigma)}{\text{cost}(\sigma)} \mid \text{cost}(\sigma) = n \right\}.$$

Let  $\mathcal{O}$  be some class of box packing algorithms (for instance online algorithms). The *optimal asymptotic performance ratio* for  $\mathcal{O}$  is defined to be  $\mathcal{R}_{\mathcal{O}}^{\infty} = \inf_{\mathcal{A} \in \mathcal{O}} \mathcal{R}_{\mathcal{A}}^{\infty}$ . Given  $\mathcal{O}$ , our goal is to find an algorithm with asymptotic performance ratio close to  $\mathcal{R}_{\mathcal{O}}^{\infty}$ . In this paper, we also consider the *resource augmented* box packing problem, where the online algorithm has larger bins at its disposal than the offline algorithm, and the goal is to minimize the (relative) number of bins used. Here all online bins are of the same size, and all the offline bins are of the same size, but these two sizes are not necessarily the same.

**Previous Results:** The classic online bin packing problem was first investigated by Ullman [17]. The lower bound currently stands at 1.54014, due to van Vliet [18]. Define

$$\pi_{i+1} = \pi_i(\pi_i - 1) + 1, \quad \pi_1 = 2,$$

and

$$H_{\infty} = \sum_{i=1}^{\infty} \frac{1}{\pi_i - 1} \approx 1.69103.$$

Lee and Lee presented an algorithm called HARMONIC, which classifies items into  $m > 1$  classes based on their size and uses bounded space. For any  $\varepsilon > 0$ , there is a number  $m$  such that the HARMONIC algorithm that uses  $m$  classes has a performance ratio of at most  $(1 + \varepsilon)H_{\infty}$  [11]. They also showed there is no bounded space algorithm with a performance ratio below  $H_{\infty}$ . Currently the best known unbounded space upper bound is 1.58889 due to Seiden [15].

The variable-sized bin packing problem was first investigated by Friesen and Langston [8]. Csirik [3] proposed the VARIABLE HARMONIC algorithm and showed that it has performance ratio at most  $H_{\infty}$ . Seiden [14] showed that this algorithm is optimal among bounded space algorithms.

The resource augmented bin packing problem was studied by Csirik and Woeginger [6]. They showed that the optimal bounded space asymptotic performance ratio is a function  $\rho(b)$  of the online bin size  $b$ .

The online hyperbox packing problem was first investigated by Coppersmith and Raghavan [2], who gave an algorithm based on NEXT FIT with performance ratio  $\frac{13}{4} = 3.25$  for  $d = 2$ . Csirik, Frenk and Labbe [4] gave an algorithm based on FIRST FIT with performance ratio  $\frac{49}{16} = 3.0625$  for  $d = 2$ . Csirik and van Vliet [5] presented an algorithm with performance ratio  $(H_{\infty})^d$  for all  $d \geq 2$  (2.85958 for  $d = 2$ ). Even though this algorithm is based on HARMONIC, it was not clear how to change it to bounded space. Li and Cheng [13] also gave a HARMONIC-based algorithm for  $d = 2$  and  $d = 3$ .

Seiden and van Stee [16] improved the upper bound for  $d = 2$  to 2.66013. Several lower bounds have been shown [9,10,19,1]. The best lower bound for  $d = 2$  is 1.907 [1], while the best lower bound for large  $d$  is less than 3. For bounded space algorithms, a lower bound of  $(H_\infty)^d$  is implied by [5]. A matching upper bound was shown in [7]. This was done by giving an extension of HARMONIC which uses only bounded space. In this paper a new technique of dealing with items that are relatively small in some of their dimensions was introduced.

**Our Results:** In this paper, we present a number of results for online multidimensional packing:

- We present a bounded space algorithm for the variable-sized multidimensional bin packing problem. An interesting feature of the analysis is that although we show the algorithm is optimal, we do not know the exact asymptotic performance ratio.
- We then give an analogous algorithm for the problem of resource augmented online bin packing. This algorithm is also optimal, with a asymptotic performance ratio of  $\prod_{i=1}^d \rho(b_i)$  where  $b_1 \times \dots \times b_d$  is the size of the online bins.
- Additionally, we give a new general lower bound for unbounded space hypercube packing, where all items to be packed are hypercubes. We improve the bounds from [16].

For the variable-sized packing, the method used in this paper generalizes and combines the methods used in [7] and [14]. There are some new ingredients that we use in this paper. An important difference with the previous papers is that in the current problem, it is no longer immediately clear which bin size should be used for any given item. In [7] all bins have the same size. In [14] bins have one dimension and therefore a simple greedy choice gives the best option. In our case, all the dimensions in which an item is “large” must be taken into account, and the process of choosing a bin is done by considering all these dimensions. This in turn changes the analysis, as not only the packing could have been done in a different way by an offline algorithm, but also items could have been assigned to totally different bins (larger in some dimensions and smaller in others).

## 2 An Optimal Algorithm for Bounded Space Variable-Sized Packing

In this section we consider the problem of multidimensional packing where the bins used can have different sizes. We assume that all bins are hypercubes, with sides  $\alpha_1 < \alpha_2 < \dots < \alpha_m = 1$ . In fact our algorithm is more general and works for the case where the bins are hyperboxes with dimensions  $\alpha_{ij}$  ( $i = 1, \dots, m, j = 1, \dots, d$ ). We present the special case of bins that are hypercubes in this paper in order to avoid an overburdened notation and messy technical details.

The main structure of the algorithm is identical to the one in [7]. The main problem in adapting that algorithm to the current problem is selecting the right bin size to pack the items in. In the one-dimensional variable-sized bin packing problem, it is easy to see which bin will accommodate any given item the best; here it is not so obvious how to select the right bin size, since in one dimension a bin of a certain size might seem best whereas for other dimensions, other bins seem more appropriate.

We begin by defining types for hyperboxes based on their components and the available bin sizes. The algorithm uses a parameter  $\varepsilon$ . The value of  $M$  as a function of  $\varepsilon$  is picked so that  $M \geq 1/(1 - (1 - \varepsilon)^{1/(d+2)}) - 1$ . An arriving hyperbox  $h$  of dimensions  $(h_1, h_2, \dots, h_d)$  is classified as one of at most  $(2mM/\alpha_1 - 1)^d$  types depending on its components: a type of a hyperbox is the vector of the types of its components. We define

$$T_i = \left\{ \frac{\alpha_i}{j} \mid j \in \mathbb{N}, \frac{\alpha_i}{j} \geq \frac{\alpha_1}{2M} \right\}, \quad T = \bigcup_{i=1}^m T_i.$$

Let the members of  $T$  be  $1 = t_1 > t_2 > \dots > t_{q'} = \alpha_1/M > \dots > t_q = \alpha_1/(2M)$ . The interval  $I_j$  is defined to be  $(t_{j+1}, t_j]$  for  $j = 1, \dots, q'$ . Note that these intervals are disjoint and that they cover  $(\alpha_1/M, 1]$ .

A component larger than  $\alpha_1/M$  has type  $i$  if  $h_i \in I_i$ , and is called large. A component smaller than  $\alpha_1/M$  has type  $i$ , where  $q' \leq i \leq q-1$ , if there exists a non-negative integer  $f_i$  such that  $t_{i+1} < 2^{f_i} h_i \leq t_i$ . Such components are called small. Thus in total there are  $q-1 \leq 2mM/\alpha_1 - 1$  component types.

*Bin selection.* We now describe how to select a bin for a given type. Intuitively, the size of this bin is chosen in order to maximize the number of items packed relative to the area used. This is done as follows.

For a given component type  $s_i$  and bin size  $\alpha_j$ , write  $F(s_i, \alpha_j) = \max\{k \mid \alpha_j/k \geq t_{s_i}\}$ . Thus for a large component,  $F(s_i, \alpha_j)$  is the number of times that a component of type  $s_i$  fits in an interval of length  $\alpha_j$ . This number is uniquely defined due to the definition of the numbers  $t_i$ . Basically, the general classification into types is too fine for any particular bin size, and we use  $F(s_i, \alpha_j)$  to get a less refined classification which only considers the points  $t_i$  of the form  $\alpha_j/k$ .

Denote by  $L$  the set of components in type  $s = (s_1, \dots, s_d)$  that are large. If  $L = \emptyset$ , we use a bin of size 1 for this type. Otherwise, we place this type in a bin of any size  $\alpha_j$  which maximizes<sup>1</sup>

$$\prod_{i \in L} \frac{F(s_i, \alpha_j)}{\alpha_j}. \quad (1)$$

Thus we do not take small components into account in this formula. Note that for a small component,  $F(s_i, \alpha_j)$  is not necessarily the same as the number of times that such a component fits into any interval of length  $\alpha_j$ . However, it is at least  $M$  for any small component.

When such a bin is opened, it is split into  $\prod_{i=1}^d F(s_i, \alpha_j)$  identical sub-bins of dimensions  $(\alpha_j/F(s_1, \alpha_j), \dots, \alpha_j/F(s_d, \alpha_j))$ . These bins are then further sub-divided into sub-bins in order to place hyperboxes in “well-fitting” sub-bins, in the manner which is described in [7].

Similarly to in that paper, the following claim can now be shown. We omit the proof here due to space constraints.

<sup>1</sup> For the case that the bins are hyperboxes instead of hypercubes, we here get the formula  $\prod_{i \in L} (F(s_i, \alpha_{ij})/\alpha_{ij})$ , and similar changes throughout the text.

*Claim.* The occupied volume in each closed bin of type  $s = (s_1, \dots, s_d)$  is at least

$$V_{s,j} = (1 - \varepsilon) \alpha_j^d \prod_{i \in L} \frac{F(s_i, \alpha_j)}{F(s_i, \alpha_j) + 1},$$

where  $L$  is the set of large components in this type and  $\alpha_j$  is the bin size used to pack this type.

We now define a weighting function for our algorithm. The weight of a hyperbox  $h$  with components  $(h_1, \dots, h_d)$  and type  $s = (s_1, \dots, s_d)$  is defined as

$$w_\varepsilon(h) = \frac{1}{1 - \varepsilon} \left( \prod_{i \notin L} h_i \right) \left( \prod_{i \in L} \frac{\alpha_j}{F(s_i, \alpha_j)} \right),$$

where  $L$  is the set of large components in this type and  $\alpha_j$  is the size of bins used to pack this type.

In order to prove that this weighting function works (gives a valid upper bound for the cost of our algorithm), we will want to modify components  $s_i$  to the smallest possible component such that  $F(s_i, \alpha_j)$  does not change. (Basically, a component will be rounded to  $\alpha_j / (F(s_i, \alpha_j) + 1)$  plus a small constant.) However, with variable-sized bins, when we modify components in this way, the algorithm might decide to pack the new hyperbox differently. (Remember that  $F(s_i, \alpha_j)$  is a “less refined” classification which does not take other bin sizes than  $\alpha_j$  into account.) To circumvent this technical difficulty, we will show first that as long as the algorithm keeps using the same bin size for a given item, the volume guarantee still holds.

For a given type  $s = (s_1, \dots, s_d)$  and the corresponding set  $L$  and bin size  $\alpha_j$ , define an *extended type*  $\text{Ext}(s_1, \dots, s_d)$  as follows: an item  $h$  is of extended type  $\text{Ext}(s_1, \dots, s_d)$  if each large component  $h_i \in (\frac{\alpha_j}{F(s_i, \alpha_j) + 1}, \frac{\alpha_j}{F(s_i, \alpha_j)}]$  and each small component  $h_i$  is of type  $s_i$ .

**Corollary 1.** *Suppose items of extended type  $\text{Ext}(s_1, \dots, s_d)$  are packed into bins of size  $\alpha_j$ . Then the occupied volume in each closed bin is at least  $V_{s,j}$ .*

*Proof.* In the proof of Claim 1, we only use that each large component  $h_i$  is contained in the interval  $(\frac{\alpha_j}{F(s_i, \alpha_j) + 1}, \frac{\alpha_j}{F(s_i, \alpha_j)}]$ . Thus the proof also works for extended types.

**Lemma 1.** *For all input sequences  $\sigma$ ,  $\text{cost}_{\text{alg}}(\sigma) \leq \sum_{h \in \sigma} w_\varepsilon(h) + O(1)$ .*

*Proof.* In order to prove the claim, it is sufficient to show that each closed bin of size  $\alpha_j$  contains items of total weight of at least  $\alpha_j^d$ . Consider a bin of this size filled with hyperboxes of type  $s = (s_1, \dots, s_d)$ . It is sufficient to consider the subsequence  $\sigma$  of the input that contains only items of this type, since all types are packed independently. This subsequence only uses bins of size  $\alpha_j$  so we may assume that *no other sizes of bins are given*. We build an input  $\sigma'$  for which both the behavior of the algorithm and the weights are the same as for  $\sigma$ , and show the claim holds for  $\sigma'$ . Let  $\delta < 1/M^3$  be a very small constant.

For a hyperbox  $h \in \sigma$  with components  $(h_1, \dots, h_d)$  and type  $s = (s_1, \dots, s_d)$ , let  $h' = (h'_1, \dots, h'_d) \in \sigma'$  be defined as follows. For  $i \notin L$ ,  $h'_i = h_i$ . For  $i \in L$ ,  $h'_i = \alpha_j / (F(s_i, \alpha_j) + 1) + \delta < \alpha_j / F(s_i, \alpha_j)$ .

Note that  $h'$  is of extended type  $\text{Ext}(s_1, \dots, s_d)$ . Since only one size of bin is given, the algorithm packs  $\sigma'$  in the same way as it packs  $\sigma$ . Moreover, according to the definition of weight above,  $h$  and  $h'$  have the same weight.

Let  $v(h)$  denote the volume of an item  $h$ . For  $h \in \sigma$ , we compute the ratio of weight and volume of the item  $h'$ . We have

$$\begin{aligned} \frac{w_\varepsilon(h')}{v(h')} &= \frac{w_\varepsilon(h)}{v(h')} = \frac{1}{1 - \varepsilon} \left( \prod_{i \notin L} h'_i \right) \left( \prod_{i \in L} \frac{\alpha_j}{F(s_i, \alpha_j)} \right) \bigg/ \prod_{i=1}^d h'_i \\ &= \frac{1}{1 - \varepsilon} \prod_{i \in L} \frac{\alpha_j}{F(s_i, \alpha_j) h'_i} > \frac{1}{1 - \varepsilon} \prod_{i \in L} \frac{F(s_i, \alpha_j) + 1}{F(s_i, \alpha_j) + M \frac{\alpha_j}{\alpha_1} \delta}. \end{aligned}$$

Here we have used in the last step that a component with a large type fits less than  $M$  times in a (one-dimensional) bin of size  $\alpha_1$ , and therefore less than  $M \frac{\alpha_j}{\alpha_1}$  times in a bin of size  $\alpha_j \geq \alpha_1$ . As  $\delta$  tends to zero, this bound approaches  $\alpha_j^d / V_{s,j}$ . We find

$$w_\varepsilon(h) \geq \alpha_j^d \frac{v(h')}{V_{s,j}} \quad \text{for all } h \in \sigma.$$

Then Corollary 1 implies that the total weight of items in a closed bin of size  $\alpha_j$  is no smaller than  $\alpha_j^d$ , which is the cost of such a bin.

Suppose the optimal solution for a given input sequence  $\sigma$  uses  $n_j$  bins of size  $\alpha_j$ . Denote the  $i$ th bin of size  $\alpha_j$  by  $B_{i,j}$ . Then

$$\frac{\sum_{h \in \sigma} w_\varepsilon(h)}{\sum_{j=1}^m \alpha_j^d n_j} = \frac{\sum_{j=1}^m \sum_{i=1}^{n_j} \sum_{h \in B_{i,j}} w_\varepsilon(h)}{\sum_{j=1}^m \alpha_j^d n_j} = \frac{\sum_{j=1}^m \sum_{i=1}^{n_j} \sum_{h \in B_{i,j}} w_\varepsilon(h)}{\sum_{j=1}^m \sum_{i=1}^{n_j} \alpha_j^d}.$$

This implies that the asymptotic worst case ratio is upper bounded by

$$\max_j \max_{X_j} \sum_{h \in X_j} w_\varepsilon(h) / \alpha_j^d, \quad (2)$$

where the second maximum is taken over all sets  $X_j$  that can be packed in a bin of size  $\alpha_j$ . Similarly to in [7], it can now be shown that this weighting function is also "optimal" in that it determines the true asymptotic performance ratio of our algorithm.

In particular, it can be shown that packing a set of hyperboxes  $X$  that have the same type vectors of large and small dimensions takes at least

$$\sum_{h \in X} \prod_{i \notin L} \frac{h_i}{\alpha_j} \bigg/ \prod_{i \in L} F(s_i, \alpha_j)$$

bins of size  $\alpha_j$ , where  $h_i$  is the  $i$ th component of hyperbox  $h$ ,  $s_i$  is the type of the  $i$ th component, and  $L$  is the set of large components (for all the hyperboxes in  $X$ ). Since

the cost of such a bin is  $\alpha_j^d$ , this means that the total cost to pack  $N'$  copies of some item  $h$  is at least  $N'w_\varepsilon(h)(1 - \varepsilon)$  when bins of this size are used. However, it is clear that using bins of another size  $\alpha_k$  does not help: packing  $N'$  copies of  $h$  into such bins would give a total cost of

$$N' \left( \prod_{i \notin L} h_i \right) \left( \prod_{i \in L} \frac{\alpha_k}{F(s_i, \alpha_k)} \right).$$

Since  $\alpha_j$  was chosen to maximize  $\prod_{i \in L} (F(s_i, \alpha_j)/\alpha_j)$ , this expression cannot be less than  $N'w_\varepsilon(h)(1 - \varepsilon)$ . More precisely, any bins that are not of size  $\alpha_j$  can be replaced by the appropriate number of bins of size  $\alpha_j$  without increasing the total cost by more than 1 (it can increase by 1 due to rounding).

This implies that our algorithm is optimal among online bounded space algorithms.

### 3 An Optimal Algorithm for Bounded Space Resource Augmented Packing

The resource augmented problem is now relatively simple to solve. In this case, the online algorithm has bins to its disposal that are hypercubes of dimensions  $b_1 \times b_2 \times \dots \times b_d$ . We can use the algorithm from [7] with the following modification: the types for dimension  $j$  are not based on intervals of the form  $(1/(i+1), 1/i]$  but rather intervals of the form  $(b_j/(i+1), b_j/i]$ .

Then, to pack items of type  $s = (s_1, \dots, s_d)$ , a bin is split into  $\prod_{i=1}^d s_i$  identical sub-bins of dimensions  $(b_1/s_1, \dots, b_d/s_d)$ , and then subdivided further as necessary.

We now find that each closed bin of type  $s = (s_1, \dots, s_d)$  is full by at least

$$(1 - \varepsilon)B \prod_{i \in L} \frac{s_i}{s_i + 1},$$

where  $L$  is the set of large components in this type, and  $B = \prod_{j=1}^d b_j$  is the volume of the online bins.

We now define the weight of a hyperbox  $h$  with components  $(h_1, \dots, h_d)$  and type  $s = (s_1, \dots, s_d)$  as

$$w_\varepsilon(h) = \frac{1}{1 - \varepsilon} \left( \prod_{i \notin L} \frac{h_i}{b_i} \right) \left( \prod_{i \in L} \frac{1}{s_i} \right),$$

where  $L$  is the set of large components in this type.

This can be shown to be valid similarly to before, and it can also be shown that items can not be packed better. However, in this case we are additionally able to give explicit bounds for the asymptotic performance ratio.

### 3.1 The Asymptotic Performance Ratio

Csirik and Woeginger [6] showed the following for the one-dimensional case.

For a given bin size  $b$ , define an infinite sequence  $T(b) = \{t_1, t_2, \dots\}$  of positive integers as follows:

$$t_1 = \lfloor 1 + b \rfloor \quad \text{and} \quad r_1 = \frac{1}{b} - \frac{1}{t_1},$$

and for  $i = 1, 2, \dots$

$$t_{i+1} = \lfloor 1 + \frac{1}{r_i} \rfloor \quad \text{and} \quad r_{i+1} = r_i - \frac{1}{t_{i+1}}.$$

Define

$$\rho(b) = \sum_{i=1}^{\infty} \frac{1}{t_i - 1}.$$

**Lemma 2.** *For every bin size  $b \geq 1$ , there exist online bounded space bin packing algorithms with worst case performance arbitrarily close to  $\rho(b)$ . For every bin size  $b \geq 1$ , the bound  $\rho(b)$  cannot be beaten by any online bounded space bin packing algorithm.*

The following lemma is proved in Csirik and Van Vliet [5] for a specific weighting function which is independent of the dimension, and is similar to a result of Li and Cheng [12]. However, the proof holds for any positive one-dimensional weighting function  $w$ . We extend it for the case where the weighting function depends on the dimension. For a one-dimensional weighting function  $w_j$  and an input sequence  $\sigma$ , define  $w_j(\sigma) = \sum_{h \in \sigma} w_j(h)$ . Furthermore define  $W_j = \sup_{\sigma} w_j(\sigma)$ , where the supremum is taken over all sequences that can be packed into a one-dimensional bin.

**Lemma 3.** *Let  $\sigma$  be a list of  $d$ -dimensional rectangles, and let  $Q$  be a packing which packs these rectangles into a  $d$ -dimensional unit cube. For each  $h \in \sigma$ , we define a new hyperbox  $h'$  as follows:  $s_j(h') = w_j(s_j(h))$  for  $1 \leq j \leq d$ . Denote the resulting list of hyperboxes by  $\sigma'$ . Then there exists a packing  $Q'$  which packs  $\sigma'$  into a cube of size  $(W_1, \dots, W_d)$ .*

*Proof.* We use a construction analogous to the one in [5]. We transform the packing  $Q = Q^0$  of  $\sigma$  into a packing  $Q^d$  of  $\sigma'$  in a cube of the desired dimensions. This is done in  $d$  steps, one for each dimension. Denote the coordinates of item  $h$  in packing  $Q^i$  by  $(x_1^i(h), \dots, x_d^i(h))$ , and its dimensions by  $(s_1^i(h), \dots, s_d^i(h))$ .

In step  $i$ , the coordinates as well as the sizes in dimension  $i$  are adjusted as follows. First we adjust the sizes and set  $s_i^i(h) = w_i(s_i(h))$  for every item  $h$ , leaving other dimensions unchanged.

To adjust coordinates, for each item  $h$  in packing  $Q^{i-1}$  we find the “left-touching” items, which is the set of items  $g$  which overlap with  $h$  in  $d - 1$  dimensions, and for which  $x_i^{i-1}(g) + s_i^{i-1}(g) = x_i^{i-1}(h)$ . We may assume that for each item  $h$ , there is either a left-touching item or  $x_i^{i-1}(h) = 0$ .

Then, for each item  $h$  that has no left-touching items, we set  $x_i^i(h) = 0$ . For all other items  $h$ , starting with the ones with smallest  $i$ -coordinate, we make the  $i$ -coordinate equal to  $\max(x_i^i(g) + s_i^i(g))$ , where the maximum is taken over the left-touching items of  $h$  in packing  $S^{i-1}$ . Note that we use the new coordinates and sizes of left-touching items in this construction, and that this creates a packing without overlap.

If in any step  $i$  the items need more than  $W_i$  room, this implies a chain of left-touching items with total size less than 1 but total weight more than  $W_i$ . From this we can find a set of one-dimensional items that fit in a bin but have total weight more than  $W_i$  (using weighting function  $w_i$ ), which is a contradiction.

As in [5], this implies immediately that the total weight that can be packed into a unit-sized bin is upper bounded by  $\prod_{i=1}^d W_i$ , which in the present case is  $\prod_{i=1}^d \rho(b_i)$ . Moreover, by extending the lower bound from [6] to  $d$  dimensions exactly as in [5], it can be seen that the asymptotic performance ratio of any online bounded space bin packing algorithm can also not be lower than  $\prod_{i=1}^d \rho(b_i)$ .

#### 4 Lower Bound for Unbounded Space Hypercube Packing

We construct a sequence of items to prove a general lower bound for hypercube packing in  $d$  dimensions. In this problem, all items to be packed are hypercubes. Take an integer  $\ell > 1$ . Let  $\varepsilon > 0$  be a number smaller than  $1/2^\ell - 1/(2^\ell + 1)$ . The input sequence is defined as follows. We use a large integer  $N$ . Let

$$\begin{aligned} x_i &= (2^{\ell+1-i} - 1)^d - (2^{\ell+1-i} - 2)^d \quad i = 1, \dots, \ell \\ x_0 &= 2^{\ell d} - (2^\ell - 1)^d \end{aligned}$$

In step 0, we let  $Nx_0$  items of size  $s_0 = (1 + \varepsilon)/(2^\ell + 1)$  arrive. In step  $i = 1, \dots, \ell$ , we let  $Nx_i$  items of size  $s_i = (1 + \varepsilon)/2^{\ell+1-i}$  arrive. For  $i = 1, \dots, \ell - 1$ , item size  $s_i$  in this sequence divides all the item sizes  $s_{i+1}, \dots, s_\ell$ . Furthermore,  $\sum_{i=0}^{\ell} s_i = (1 + \varepsilon)(1 - 1/2^\ell + 1/(2^\ell + 1)) < 1$  by our choice of  $\varepsilon$ .

The online algorithm receives steps  $0, \dots, k$  of this input sequence for some (unknown)  $0 \leq k \leq \ell$ .

A pattern is a multiset of items that fits (in some way) in a unit bin. A pattern is *dominant* if when we increase the number of items of the smallest size in that pattern, the resulting multiset no longer fits in a unit bin. A pattern is *greedy* if the largest item in it appears as many times as it can fit in a bin, and this also holds for all smaller items, each time taking the larger items that are in the pattern into account. Note that not all possible sizes of items need to be present in the bin.

For a pattern  $P$  of items that all have sizes in  $\{s_0, \dots, s_\ell\}$ , denote the number of items of size  $s_i$  by  $P_i$ .

**Lemma 4.** For any pattern  $P$  for items with sizes in  $\{s_0, \dots, s_\ell\}$ ,

$$P_i \leq (2^{\ell+1-i} - 1)^d - \sum_{j=i+1}^{\ell} 2^{(j-i)d} P_j \quad i = 1, \dots, \ell, \quad P_0 \leq 2^{\ell d} - \sum_{j=1}^{\ell} 2^{(j-1)d} P_j \quad (3)$$



*Proof.* Suppose (3) does not hold for some  $i$ , and consider the smallest  $i$  for which it does not hold. Consider an item of size  $s_j$  with  $j > i$  that appears in  $P$ . If there is no such  $j$ , we have a contradiction, since at most  $(2^{\ell+1-i} - 1)^d$  items of size  $s_i$  fit in the unit hypercube for  $i = 1, \dots, \ell$ , or at most  $2^{\ell d}$  items of size  $s_0$ . This follows from Claim 4 in [7].

First suppose  $i > 0$ . If we replace an item of size  $s_j$  with  $2^{(j-i)d}$  items of size  $s_i$ , the resulting pattern is still feasible: all the new size  $s_i$  items can be placed inside the hypercube that this size  $s_j$  item has vacated.

We can do this for all items of size  $s_j$ ,  $j > i$  that appear in the pattern. This results in a pattern with only items of size  $s_i$  or smaller. Since every size  $s_j$  item is replaced by  $2^{(j-i)d}$  items of size  $s_i$ , the final pattern has more than  $(2^{\ell+1-i} - 1)^d$  items of size  $s_i$ , a contradiction.

Now suppose  $i = 0$ . In this case  $\lfloor (2^\ell + 1)/2^{\ell+1-j} \rfloor = 2^{j-1}$  items of size  $s_0$  fit in a hypercube of size  $s_j$ , and the proof continues analogously.

We define a *canonical packing* for dominant patterns. We create a grid in the bin as follows. One corner of the bin is designated as the origin  $O$ . We assign a coordinate system to the bin, where each positive axis is along some edge of the bin. The grid points are those points inside the bin that have all coordinates of the form  $m_1(1 + \varepsilon)/2^{m_2}$  for  $m_1, m_2 \in \mathbb{N} \cup \{0\}$ .

We pack the items in order of decreasing size. Each item of size  $s_i$  ( $i = 1, \dots, \ell$ ) is placed at the available grid point that has all coordinates smaller than  $1 - s_i$ , all coordinates equal to a multiple of  $s_i$  and is closest to the origin. So the first item is placed in the origin. Each item of size  $s_0$  is placed at the available grid point that has all coordinates equal to a multiple of  $s_1$  (and not  $s_0$ ) and is closest to the origin. Note that for these items we also use grid points with some coordinates equal to  $(2^\ell - 1)(1 + \varepsilon)/2^\ell$ , unlike for items of size  $s_1$ . This is feasible because  $(2^\ell - 1)(1 + \varepsilon)/2^\ell + (1 + \varepsilon)/(2^\ell + 1) = (1 + \varepsilon)(1 - 1/2^\ell + 1/(2^\ell + 1)) < 1$ .

In each step  $i$  we can place a number of items which is equal to the upper bound in Lemma 4. For  $i = 1, \dots, \ell$ , this is because a larger item of size  $s_j$  takes away exactly  $2^{(j-i)d}$  grid points that are multiples of  $s_i$ , and originally there are  $(2^{\ell+1-i} - 1)^d$  grid points of this form that have all coordinates smaller than  $1 - s_i$ . For  $i = 0$ ,  $2^{(j-1)d}$  grid points that are multiples of  $s_1$  are taken away by an item of size  $s_j$ , from an original supply of  $2^{\ell d}$ . This shows that all patterns can indeed be packed in canonical form.

**Lemma 5.** *For this set of item sizes, any dominant pattern that is not greedy is a convex combination of dominant patterns that are greedy.*

*Proof.* We use induction to construct a convex combination of greedy patterns for a given dominant pattern  $P$ . The induction hypothesis is as follows: the vector that describes the numbers of items of the  $t$  smallest types which appear in the pattern is a convex combination of greedy vectors for these types. Call such a pattern  $t$ -greedy.

The base case is  $t = 1$ . We consider the items of the smallest type that occurs in  $P$ . Since  $P$  is dominant, for this type we have that as many items as possible appear in  $P$ , given the larger items. Thus  $P$  is 1-greedy.

We now prove the induction step. Suppose that in  $P$ , items of type  $i$  appear fewer times than they could, given the larger items. Moreover,  $P$  contains items of some smaller type. Let  $i'$  be the largest smaller type in  $P$ . By induction, we only need to consider

patterns in which all the items of type less than  $i$  that appear, appear as many times as possible, starting with items of type  $i'$ . (All other patterns are convex combinations of such patterns.)

We define two patterns  $P'$  and  $P''$  such that  $P$  is a convex combination of them. First suppose  $i' > 0$ .  $P'$  is defined as follows: modify  $P$  by removing all items  $i$  and adding the largest smaller item that appears in  $P$ , of type  $i'$ ,  $2^{(i-i')d}$  times per each item  $i$ . When creating  $P'$ , we thus add the maximum amount of items of type  $i'$  that can fit for each removed item of type  $i$ .  $P$  is greedy with respect to all smaller items, and  $s_{i'}$  divides  $s_i$ . Therefore the multiset  $P'$  defined in this way is a pattern, and is  $(t+1)$ -greedy.

$P''$  on the other hand is created by adding items of phase  $i$  to  $P$  and removing items of type  $i'$ . In particular, in the canonical packing for  $P$ , at each grid point for type  $i$  that is not removed due to a higher-type item, we place an item of size  $s_i$  and remove all items that overlap with this item. Since all items smaller than  $s_i$  appear as many times as possible given the larger items, all the removed items are of the next smaller type  $i'$  that appear in  $P$ . This holds because the items are packed in order of decreasing size, so this area will certainly be filled with items of type  $i'$  if the item of size  $i$  is not there.

In  $P''$ , the number of items of type  $i$  is now maximized given items of higher types. Only type  $i'$  items are removed, and only enough to make room for type  $i$ , so type  $i'$  remains greedy. Thus  $P''$  is  $(t+1)$ -greedy. Each time that we add an item  $i$ , we remove exactly  $2^{(i-i')d}$  items of type  $i'$ . So by adding an item  $i$  in creating  $P''$ , we remove exactly the same number of items of type  $i'$  as we add when we remove an item  $i$  while creating  $P'$ . Therefore,  $P$  is a convex combination of  $P'$  and  $P''$ , and we are done.

Now suppose  $i' = 0$ . (This is a special case of the induction step for  $t+1 = 2$ .) In this case, in  $P'$  each item of type  $i$  is replaced by  $2^{(i-1)d}$  items of type 0. In the canonical packing, this is exactly the number of type 1 grid points that become available when removing an item of type  $i$ . Thus in  $P'$ , the number of type 0 items is maximal (since it is sufficient to use type 1 grid points for them), and  $P'$  is 2-greedy.

Similarly, it can be seen that to create  $P''$ , we need to remove  $2^{(i-1)d}$  items of type 0 in the canonical packing in order to place each item of type  $i$ . Then  $P''$  is 2-greedy, and again  $P$  is a convex combination of  $P'$  and  $P''$ .

We can now formulate a linear program to lower bound the asymptotic performance ratio of any unbounded space online algorithm as in [16]. We will need the offline cost to pack any prefix of the full sequence. This is calculated as follows.

To pack the items of the largest type  $k$ , which have size  $(1+\varepsilon)2^{k-\ell-1}$ , we need  $Nx_k/(2^{\ell+1-k} - 1)^d$  bins because there are  $Nx_k$  such items. These are all packed identically: using a canonical packing, we pack as many items of smaller types in bins with these items as possible. (Thus we use a greedy pattern.) Some items of types  $1, \dots, k-1$  still remain to be packed. It is straightforward to calculate the number of items of type  $k-1$  that still need to be packed, and how many bins this takes. We continue in the same manner until all items are packed.

Solving this linear program for  $\ell = 11$  and several values of  $d$  gives us the following results. The first row contains the previously known lower bounds.

$d$	1	2	3	4	5	6	7	8	9	10
old	1.5401	1.6217	1.60185	1.5569	(Bounds here were below 1.5569)					
new	(1.5)	1.6406	1.6680	1.6775	1.6840	1.6887	1.6920	1.6943	1.6959	1.6973

References

1. David Blitz, André van Vliet, and Gerhard J. Woeginger. Lower bounds on the asymptotic worst-case ratio of online bin packing algorithms. Unpublished manuscript, 1996.

2. Don Coppersmith and Prabhakar Raghavan. Multidimensional online bin packing: Algorithms and worst case analysis. *Operations Research Letters*, 8:17–20, 1989.

3. János Csirik. An online algorithm for variable-sized bin packing. *Acta Informatica*, 26:697–709, 1989.

4. János Csirik, Johannes B. G. Frenk, and Martine Labbe. Two dimensional rectangle packing: on line methods and results. *Discrete Applied Mathematics*, 45:197–204, 1993.

5. János Csirik and André van Vliet. An on-line algorithm for multidimensional bin packing. *Operations Research Letters*, 13(3):149–158, Apr 1993.

6. János Csirik and Gerhard J. Woeginger. Resource augmentation for online bounded space bin packing. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 296–304, Jul 2000.

7. Leah Epstein and Rob van Stee. Optimal online bounded space multidimensional packing. In *Proc. of 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, pages 207–216. ACM/SIAM, 2004.

8. D. K. Friesen and M. A. Langston. Variable sized bin packing. *SIAM Journal on Computing*, 15:222–230, 1986.

9. Gabor Galambos. A 1.6 lower bound for the two-dimensional online rectangle bin packing. *Acta Cybernetica*, 10:21–24, 1991.

10. Gabor Galambos and André van Vliet. Lower bounds for 1-, 2-, and 3-dimensional online bin packing algorithms. *Computing*, 52:281–297, 1994.

11. C. C. Lee and D. T. Lee. A simple online bin packing algorithm. *Journal of the ACM*, 32:562–572, 1985.

12. K. Li and K. H. Cheng. Generalized First-Fit algorithms in two and three dimensions. *International Journal on Foundations of Computer Science*, 1(2):131–150, 1990.

13. Keqin Li and Kam-Hoi Cheng. A generalized harmonic algorithm for on-line multi-dimensional bin packing. Technical Report UH-CS-90-2, University of Houston, January 1990.

14. Steve S. Seiden. An optimal online algorithm for bounded space variable-sized bin packing. *SIAM Journal on Discrete Mathematics*, 14(4):458–470, 2001.

15. Steve S. Seiden. On the online bin packing problem. *Journal of the ACM*, 49(5):640–671, 2002.

16. Steve S. Seiden and Rob van Stee. New bounds for multi-dimensional packing. *Algorithmica*, 36(3):261–293, 2003.

17. Jeffrey D. Ullman. The performance of a memory allocation algorithm. Technical Report 100, Princeton University, Princeton, NJ, 1971.

18. André van Vliet. An improved lower bound for online bin packing algorithms. *Information Processing Letters*, 43:277–284, 1992.

19. André van Vliet. *Lower and upper bounds for online bin packing and scheduling heuristics*. PhD thesis, Erasmus University, Rotterdam, The Netherlands, 1995.

# An Inductive Construction for Plane Laman Graphs via Vertex Splitting

Zsolt Fekete<sup>1,2</sup>, Tibor Jordán<sup>1</sup>, and Walter Whiteley<sup>3</sup>

<sup>1</sup> Department of Operations Research, Eötvös University, Pázmány sétány 1/C, 1117 Budapest, Hungary [jordan@cs.elte.hu](mailto:jordan@cs.elte.hu) <sup>†</sup>

<sup>2</sup> Communication Networks Laboratory, Pázmány sétány 1/A, 1117 Budapest, Hungary [fezso@cs.elte.hu](mailto:fezso@cs.elte.hu)

<sup>3</sup> Department of Mathematics and Statistics, York University, Toronto, Canada [whiteley@mathstat.yorku.ca](mailto:whiteley@mathstat.yorku.ca)

**Abstract.** We prove that all planar Laman graphs (i.e. minimally generically rigid graphs with a non-crossing planar embedding) can be generated from a single edge by a sequence of vertex splits. It has been shown recently [6,12] that a graph has a pointed pseudo-triangular embedding if and only if it is a planar Laman graph. Due to this connection, our result gives a new tool for attacking problems in the area of pseudo-triangulations and related geometric objects. One advantage of vertex splitting over alternate constructions, such as edge-splitting, is that vertex splitting is geometrically more local.

We also give new inductive constructions for duals of planar Laman graphs and for planar generically rigid graphs containing a unique rigidity circuit. Our constructions can be found in  $O(n^3)$  time, which matches the best running time bound that has been achieved for other inductive constructions.

## 1 Introduction

The characterization of graphs for rigidity circuits in the plane and isostatic graphs in the plane has received significant attention in the last few years [2, 3,8,9,14]. The special case of planar graphs, and their non-crossing realizations has been a particular focus, in part because special inductive constructions apply [3], and in part because of special geometric realizations as pseudo-triangulations and related geometric objects [6,11,12,13].

In [3] it was observed that all 3-connected planar rigidity circuits can be generated from  $K_4$  by a sequence of vertex splits, preserving planarity, 3-connectivity and the circuit property in every intermediate step, a result that follows by duality from the construction of 3-connected planar rigidity circuits by edge splits [2]. We extend this result in two ways. We show that all planar Laman graphs (bases for the rigidity matroid) can be generated from  $K_2$  (a single edge) by a

---

<sup>†</sup> Supported by the MTA-ELTE Egerváry Research Group on Combinatorial Optimization, and the Hungarian Scientific Research Fund grant no. F034930, T037547, and FKFP grant no. 0143/2001.

sequence of vertex splits, and all planar generically rigid graphs (spanning sets of the rigidity matroid) containing a unique rigidity circuit can be generated from  $K_4$  (a complete graph on four vertices) by a sequence of vertex splits.

One advantage of vertex splitting over alternate constructions, such as edge-splitting, is that vertex splitting is geometrically more local. With very local moves, one can better ensure the planarity of a class of realizations of the resulting graph. This feature can be applied to give an alternate proof that each planar Laman graph can be realized as a pointed pseudo-triangulation, and that each planar generically rigid graph with a unique rigidity circuit can be realized as a pseudo-triangulation with a single non-pointed vertex.

A graph  $G = (V, E)$  is a *Laman graph* if  $|V| \geq 2$ ,  $|E| = 2|V| - 3$ , and

$$i(X) \leq 2|X| - 3 \quad (1)$$

holds for all  $X \subseteq V$  with  $|X| \geq 2$ , where  $i(X)$  denotes the number of edges induced by  $X$ . Laman graphs, also known as isostatic or generically minimally rigid graphs, play a key role in 2-dimensional rigidity, see [5,7,8,10,14,16]. By Laman's Theorem [9] a graph embedded on a generic set of points in the plane is infinitesimally rigid if and only if it is Laman. Laman graphs correspond to bases of the 2-dimensional *rigidity matroid* [16] and occur in a number of geometric problems (e.g. unique realizability [8], straightening polygonal linkages [12], etc.)

Laman graphs have a well-known inductive construction, called *Henneberg construction*. Starting from an edge (a Laman graph on two vertices), construct a graph by adding new vertices one by one, by using one of the following two operations:

- (i) add a new vertex and connect it to two distinct old vertices via two new edges (*vertex addition*)
- (ii) remove an old edge, add a new vertex, and connect it to the endvertices of the removed edge and to a third old vertex which is not incident with the removed edge (*edge splitting*)

It is easy to check that a graph constructed by these operations is Laman. The more difficult part is to show that every Laman graph can be obtained this way.

**Theorem 1.** [14,8]. *A graph is Laman if and only if it has a Henneberg construction.*

An embedding  $G(P)$  of the graph  $G$  on a set of points  $P = \{p_1, \dots, p_n\} \subset \mathbb{R}^2$  is a mapping of vertices  $v_i \in V$  to points  $p_i \in P$  in the Euclidean plane. The edges are mapped to straight line segments. Vertex  $v_i$  of the embedding  $G(P)$  is *pointed* if all its adjacent edges lie on one side of some line through  $p_i$ . An embedding  $G(P)$  is *non-crossing* if no pair of segments, corresponding to independent edges of  $G$ , have a point in common, and segments corresponding to adjacent edges have only their endvertices in common. A graph  $G$  is *planar* if it has a non-crossing embedding. By a *plane* graph we mean a planar graph together with a non-crossing embedding.

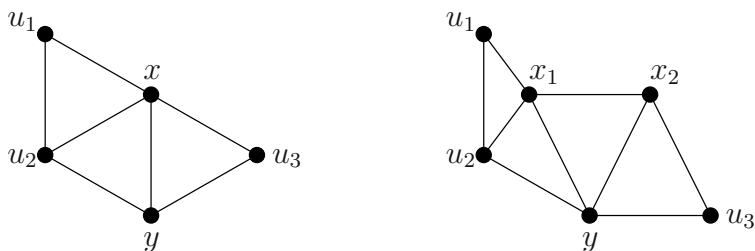
A *pseudo-triangle* is a simple planar polygon with exactly three convex vertices. A *pseudo-triangulation* of a planar set of points  $P$  is a non-crossing embedded graph  $G(P)$  whose outer face is convex and all interior faces are pseudo-triangles. In a *pointed pseudo-triangulation* all vertices are pointed. A *pointed-plus-one pseudo-triangulation* is a pseudo-triangulation with exactly one non-pointed vertex. The following result, due to Streinu, generated several exciting questions in computational geometry.

**Theorem 2.** [12] *Let  $G$  be embedded on the set  $P = \{p_1, \dots, p_n\}$  of points. If  $G$  is a pointed pseudo-triangulation of  $P$  then  $G$  is a planar Laman graph.*

One of the first questions has been answered by the next theorem. The proof used the following topological version of the Henneberg construction, which is easy to deduce from Theorem 1.

**Lemma 1.** *Every plane Laman graph has a plane Henneberg construction, where all intermediate graphs are plane, and at each step the topology is changed only on edges and faces involved in the Henneberg step. In addition, if the outer face of the plane graph is a triangle, there is a Henneberg construction starting from that triangle.*

**Theorem 3.** [6] *Every planar Laman graph can be embedded as a pointed pseudo-triangulation.*



**Fig. 1.** The plane vertex splitting operation applied to a plane Laman graph on edge  $xy$  with partition  $E_1 = \{xu_1, xu_2\}$ ,  $E_2 = \{xu_3\}$ .

Our main theorem is a different inductive construction for plane Laman graphs. It can also be used to prove Theorem 3 and it might be more suitable than the Henneberg construction for certain geometric problems. The construction uses (the topological version of) vertex splittings, called *plane vertex splitting*. This operation picks an edge  $xy$  in a plane graph, partitions the edges incident to  $x$  (except  $xy$ ) into two consecutive sets  $E_1, E_2$  of edges (with respect to the natural cyclic ordering given by the embedding), replaces  $x$  by two vertices  $x_1, x_2$ , attaches the edges in  $E_1$  to  $x_1$ , attaches the edges in  $E_2$  to  $x_2$ , and

adds the edges  $yx_1, yx_2, x_1x_2$ , see Figure 1. It is easy to see that plane vertex splitting, when applied to a plane Laman graph, yields a plane Laman graph. Note that the standard version of vertex splitting (see [15,16] for its applications in rigidity theory), where the partition of the edges incident to  $x$  is arbitrary, preserves the property of being Laman, but may destroy planarity.

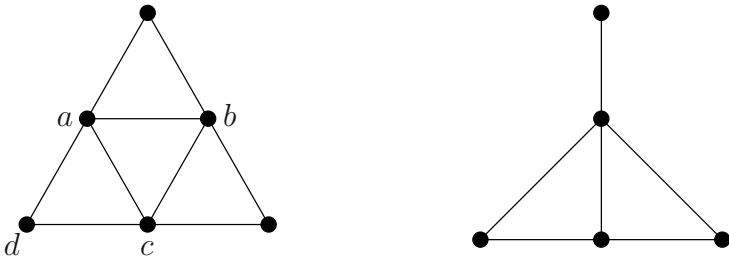
We shall prove that every plane Laman graph can be obtained from an edge by plane vertex splittings. To prove this we need to show that the inverse operation of plane vertex splitting can be performed on every plane Laman graph on at least three vertices in such a way that the graph remains (plane and) Laman. The inverse operation contracts an edge of a triangle face.

Let  $e = uv$  be an edge of  $G$ . The operation *contracting* the edge  $e$  identifies the two end-vertices of  $e$  and deletes the resulting loop as well as one edge from each of the resulting pairs of parallel edges, if there exist any. The graph obtained from  $G$  by contracting  $e$  is denoted by  $G/e$ . We say that  $e$  is *contractible* in a Laman graph  $G$  if  $G/e$  is also Laman. Observe that by contracting an edge  $e$  the number of vertices is decreased by one, and the number of edges is decreased by the number of triangles that contain  $e$  plus one. Thus a contractible edge belongs to exactly one triangle of  $G$ .

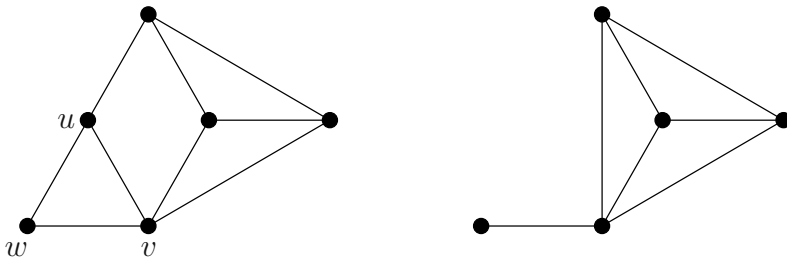
Laman graphs in general do not necessarily contain triangles, see e.g.  $K_{3,3}$ . For plane Laman graphs we can use Euler's formula to deduce the following.

**Lemma 2.** *Every plane Laman graph  $G = (V, E)$  with  $|V| \geq 4$  contains at least two triangle faces (with distinct boundaries).*

It is easy to observe that an edge of a triangle face of a plane Laman graph is not necessarily contractible. In addition, a triangle face may contain no contractible edges at all. See Figure 2 and Figure 3 for examples. This is one reason why the proof of the inductive construction via vertex splitting is more difficult than that of Theorem 1, where the corresponding inverse operations of vertex addition or edge splitting can be performed at *every* vertex of degree two or three.



**Fig. 2.** A Laman graph  $G$  and a non-contractible edge  $ab$  on a triangle face  $abc$ . The graph obtained by contracting  $ab$  satisfies (1), but it has less edges than it should have. No edge on  $abc$  is contractible, but edges  $ad$  and  $cd$  are contractible in  $G$ .



**Fig. 3.** A Laman graph  $G$  and a non-contractible edge  $uv$  on a triangle face  $uvw$ . The graph obtained by contracting  $uv$  has the right number of edges but it violates (1).

The paper is organised as follows. In Section 2 we prove some basic facts about Laman graphs. In Section 3 we turn to plane Laman graphs and complete the proof of our main result. Sections 4 and 5 contain some corollaries concerning pseudo-triangulations and further inductive constructions obtained by planar duality. Algorithmic issues are discussed in Section 6.

## 2 Preliminary Results on Laman Graphs

In this section we introduce the basic definitions and notation, and prove some preliminary lemmas on (not necessarily planar) Laman graphs.

Let  $G = (V, E)$  be a graph. For a subset  $X \subseteq V$  let  $G[X]$  denote the subgraph of  $G$  induced by  $X$ . For a pair  $X, Y \subseteq V$  let  $d_G(X, Y)$  denote the number of edges from  $X - Y$  to  $Y - X$ . We omit the subscript if the graph is clear from the context. The following equality is well-known. It is easy to check by counting the contribution of an edge to each of the two sides.

**Lemma 3.** *Let  $G = (V, E)$  be a graph and let  $X, Y \subseteq V$ . Then*

$$i(X) + i(Y) + d(X, Y) = i(X \cup Y) + i(X \cap Y).$$

Let  $G = (V, E)$  be a Laman graph. It is easy to see that Laman graphs are *simple* (i.e they contain no multiple edges) and 2-vertex-connected. A set  $X \subseteq V$  is *critical* in  $G$  if  $i(X) = 2|X| - 3$ , that is, if  $X$  satisfies (1) with equality. Equivalently,  $X$  is critical if and only if  $G[X]$  is Laman. Thus  $V$  as well as the set  $\{u, v\}$  for all  $uv \in E$  are critical. The next lemma is easy to deduce from Lemma 3.

**Lemma 4.** *Let  $G = (V, E)$  be a Laman graph and let  $X, Y \subseteq V$  be critical sets in  $G$  with  $|X \cap Y| \geq 2$ . Then  $X \cap Y$  and  $X \cup Y$  are also critical and  $d(X, Y) = 0$  holds.*

**Lemma 5.** *Let  $G = (V, E)$  be a Laman graph and let  $X \subseteq V$  be a critical set. Let  $C$  be the union of some of the connected components of  $G - X$ . Then  $X \cup C$  is critical.*



*Proof.* Let  $C_1, C_2, \dots, C_k$  be the connected components of  $G - X$  and let  $X_i = X \cup C_i$ , for  $1 \leq i \leq k$ . We have  $X_i \cap X_j = X$  and  $d(X_i, X_j) = 0$  for all  $1 \leq i < j \leq k$ , and  $\cup_{i=1}^k X_i = V$ . Since  $G$  is Laman and  $X$  is critical, we can count as follows:  $2|V| - 3 = |E| = i(X_1 \cup X_2 \cup \dots \cup X_k) = \sum_1^k i(X_i) - (k-1)i(X) \leq \sum_1^k (2|X_i| - 3) - (k-1)(2|X| - 3) = 2 \sum_1^k |X_i| + 2(k-1)|X| - 3k + 3(k-1) = 2|V| - 3$ . Thus equality must hold everywhere, and hence each  $X_i$  is critical.

Now Lemma 4 (and the fact that  $|X| \geq 2$ ) implies that if  $C$  is the union of some of the components of  $G - X$  then  $X \cup C$  is critical.  $\square$

The next lemma characterises the contractible edges in a Laman graph.

**Lemma 6.** *Let  $G = (V, E)$  be a Laman graph and let  $e = uv \in E$ . Then  $e$  is contractible if and only if there is a unique triangle  $uvw$  in  $G$  containing  $e$  and there exists no critical set  $X$  in  $G$  with  $u, v \in X$ ,  $w \notin X$ , and  $|X| \geq 4$ .*

*Proof.* First suppose that  $e$  is contractible. Then  $G/e$  is Laman, and, as we noted earlier,  $e$  must belong to a unique triangle  $uvw$ . For a contradiction suppose that  $X$  is a critical set with  $u, v \in X$ ,  $w \notin X$ , and  $|X| \geq 4$ . Then  $e$  is an edge of  $G[X]$  but it does not belong to any triangle in  $G[X]$ . Hence by contracting  $e$  we decrease the number of vertices and edges in  $G[X]$  by one. This would make the vertex set of  $G[X]/e$  violate (1) in  $G/e$ . Thus such a critical set cannot exist.

To see the ‘if’ direction suppose that there is a unique triangle  $uvw$  in  $G$  containing  $e$  and there exists no critical set  $X$  in  $G$  with  $u, v \in X$ ,  $w \notin X$ , and  $|X| \geq 4$ . For a contradiction suppose that  $G' := G/e$  is not Laman. Let  $v'$  denote the vertex of  $G'$  obtained by contracting  $e$ . Since  $G$  is Laman and  $e$  belongs to exactly one triangle in  $G$ , it follows that  $|E(G')| = 2|V(G')| - 3$ , so there is a set  $Y \subset V(G')$  with  $|Y| \geq 2$  and  $i_{G'}(Y) \geq 2|Y| - 2$ . Since  $G'$  is simple and  $uv$  belongs to a unique triangle in  $G$ , it follows that  $V(G')$ , all two-element subsets of  $V(G')$ , all subsets containing  $v'$  and  $w$ , as well as all subsets not containing  $v'$  satisfy (1) in  $G'$ . Thus we must have  $|Y| \geq 3$ ,  $v' \in Y$  and  $w \notin Y$ . Hence  $X := (Y - v') \cup \{u, v\}$  is a critical set in  $G$  with  $u, v \in X$ ,  $w \notin X$ , and  $|X| \geq 4$ , a contradiction. This completes the proof of the lemma.  $\square$

Thus two kinds of substructures can make an edge  $e = uv$  of a triangle  $uvw$  non-contractible: a triangle  $uvw'$  with  $w' \neq w$  and a critical set  $X$  with  $u, v \in X$ ,  $w \notin X$  and  $|X| \geq 4$ . Since a triangle is also critical, these substructures can be treated simultaneously. We say that a critical set  $X \subset V$  is a *blocker* of edge  $e = uv$  (with respect to the triangle  $uvw$ ) if  $u, v \in X$ ,  $w \notin X$  and  $|X| \geq 3$ .

**Lemma 7.** *Let  $uvw$  be a triangle in a Laman graph  $G = (V, E)$  and suppose that  $e = uv$  is non-contractible. Then there exists a unique maximal blocker  $X$  of  $e$  with respect to  $uvw$ . Furthermore,  $G - X$  has precisely one connected component.*

*Proof.* There is a blocker of  $e$  with respect to  $uvw$  by Lemma 6. By Lemma 4 the union of two blockers of  $e$ , with respect to  $uvw$ , is also a blocker with respect to  $uvw$ . This proves the first assertion. The second one follows from Lemma 5: let  $C$  be the union of those components of  $G - X$  that do not contain  $w$ , where  $X$  is the

maximal blocker of  $e$  with respect to  $uvw$ . Since  $X \cup C$  is critical, and does not contain  $w$ , it is also a blocker of  $e$  with respect to  $uvw$ . By the maximality of  $X$  we must have  $C = \emptyset$ . Thus  $G - X$  has only one component (which contains  $w$ ).  $\square$

Since a blocker  $X$  is a critical set in  $G$ ,  $G[X]$  is also Laman.

**Lemma 8.** *Let  $G = (V, E)$  be a Laman graph, let  $uvw$  be a triangle, and let  $f = uv$  be a non-contractible edge. Let  $X$  be the maximal blocker of  $f$  with respect to  $uvw$ . If  $e \neq f$  is contractible in  $G[X]$  then it is contractible in  $G$ .*

*Proof.* Let  $e = rz$ . Since  $e$  is contractible in  $G[X]$ , there exists a unique triangle  $rzy$  in  $G[X]$  which contains  $e$ . For a contradiction suppose that  $e$  is not contractible in  $G$ . Then by Lemma 6 there exists a blocker of  $e$  with respect to  $rzy$ , that is, a critical set  $Z \subset V$  with  $r, z \in Z$ ,  $y \notin Z$ , and  $|Z| \geq 3$ . Lemma 4 implies that  $Z \cap X$  is critical. If  $|Z \cap X| \geq 3$  then  $Z \cap X$  is a blocker of  $e$  in  $G[X]$ , contradicting the fact that  $e$  is contractible in  $G[X]$ .

Thus  $Z \cap X = \{r, z\}$ . We claim that  $w \notin Z$ . To see this suppose that  $w \in Z$  holds. Then  $w \in Z - X$ . Since  $e \neq f$ , and  $|Z \cap X| = 2$ , at least one of  $u, v$  is not in  $Z$ . But this would imply  $d(X, Z) \geq 1$ , contradicting Lemma 4. This proves  $w \notin Z$ .

Clearly,  $Z - X \neq \emptyset$ . Thus, since  $Z \cup X$  is critical by Lemma 4, it follows that  $Z \cup X$  is a blocker of  $f$  in  $G$  with respect to  $uvw$ , contradicting the maximality of  $X$ . This proves the lemma.  $\square$

### 3 Plane Laman Graphs

**Lemma 9.** *Let  $G = (V, E)$  be a plane Laman graph, let  $uvw$  be a triangle face, and let  $f = uv$  be a non-contractible edge. Let  $X$  be the maximal blocker of  $f$  with respect to  $uvw$ . Then all but one faces of  $G[X]$  are faces of  $G$ .*

*Proof.* Consider the faces of  $G[X]$  and the connected component  $C$  of  $G - X$ , which is unique by Lemma 7. Clearly,  $C$  is within one of the faces of  $G[X]$ . Thus all faces, except the one which has  $w$  in its interior, is a face of  $G$ , too.  $\square$

The exceptional face of  $G[X]$  (which is not a face of  $G$ ) is called the *special face* of  $G[X]$ . Since the special face has  $w$  in its interior, and  $uvw$  is a triangle face in  $G$ , it follows that the edge  $uv$  is on the boundary of the special face. If the special face of  $G[X]$  is a triangle  $uvq$ , then the third vertex  $q$  of this face is called the *special vertex* of  $G[X]$ . If the special face of  $G[X]$  is not a triangle, then  $X$  is a *nice blocker*. We say that an edge  $e$  is *face contractible* in a plane Laman graph if  $e$  is contractible and the triangle containing  $e$  (which is unique by Lemma 6) is a face in the given embedding.

We are ready to prove our main result on the existence of a face contractible edge. In fact, we shall prove a somewhat stronger result which will also be useful when we prove some extensions later.

**Theorem 4.** *Let  $G = (V, E)$  be a plane Laman graph with  $|V| \geq 4$ . Then*

*(i) if  $uvw$  is a triangle face,  $f = uv$  is not contractible, and  $X$  is the maximal blocker of  $f$  with respect to  $uvw$ , then there is an edge in  $G[X]$  which is face contractible in  $G$ ,*

*(ii) for each vertex  $r \in V$  there exist at least two face contractible edges which are not incident with  $r$ .*

*Proof.* The proof is by induction on  $|V|$ . It is easy to check that the theorem holds if  $|V| = 4$  (in this case  $G$  is unique and has essentially one possible planar embedding). So let us suppose that  $|V| \geq 5$  and the theorem holds for graphs with less than  $|V|$  vertices.

First we prove (i). Consider a triangle face  $uvw$  for which  $f = uv$  is not contractible, and let  $X$  be the maximal blocker of  $f$  with respect to  $uvw$ . Since  $X$  is a critical set, the induced subgraph  $G[X]$  is Laman. Together with the embedding obtained by restricting the embedding of  $G$  to the vertices and edges of its subgraph induced by  $X$ , the graph  $G[X]$  is a plane Laman graph. Since  $w \notin X$ ,  $G[X]$  has less than  $|V|$  vertices.

We call an edge  $e$  of  $G[X]$  *proper* if  $e \neq f$ ,  $e$  is face contractible in  $G[X]$ , and the triangle face of  $G[X]$  containing  $e$  is a face of  $G$  as well. It follows from the definition and Lemma 8 that a proper edge  $e$  is face contractible in  $G$  as well. We shall prove (i) by showing that there is a proper edge in  $G[X]$ .

To this end first suppose that  $|X| = 3$ . Then  $G[X]$  is a triangle, and each of its edges is contractible in  $G[X]$ . By Lemma 9 one of the two faces of  $G[X]$  is a face of  $G$  as well. Thus each of the two edges of  $G[X]$  which are different from  $f$ , is proper.

Next suppose that  $|X| \geq 4$ . By the induction hypothesis (ii) holds for  $G[X]$  by choosing  $r = u$ . Thus there exist two face contractible edges  $e', e''$  in  $G[X]$  which are not incident with  $u$  (and hence  $e'$  and  $e''$  must be different from  $f$ ). If  $X$  is a nice blocker then the triangle face containing  $e'$  (or  $e''$ ) in  $G[X]$  is a face of  $G$  as well, by Lemma 9. Thus  $e'$  (or  $e''$ ) is proper.

If  $X$  is not a nice blocker then it has a special triangle face  $uvq$ , which is not a face of  $G$ , and each of the other faces of  $G[X]$  is a face of  $G$  by Lemma 9. Since  $e'$  and  $e''$  are distinct edges which are not incident with  $u$ , at least one of them, say  $e'$ , is not an edge of the triangle  $uvq$ . Hence the triangle face of  $G[X]$  containing  $e'$  is a triangle face of  $G$  as well. Thus  $e'$  is proper. This completes the proof of (i).

It remains to prove (ii). To this end let us fix a vertex  $r \in V$ . We have two cases to consider.

**Case 1.** There exists a triangle face  $uvw$  in  $G$  with  $r \notin \{u, v, w\}$ .

If at least two edges on the triangle face  $uvw$  are face contractible then we are done. Otherwise we have blockers for two or three edges of  $uvw$ .

If none of the edges of the triangle  $uvw$  is contractible then there exist maximal blockers  $X, Y, Z$  for the edges  $vw, uw$ , and  $uv$  (with respect to  $u, v$ , and  $w$ ), respectively. By Lemma 4 we must have  $X \cap Y = \{w\}$ ,  $X \cap Z = \{v\}$ , and  $Y \cap Z = \{u\}$  (since the sets are critical and  $d(Y, Z), d(X, Y), d(X, Z) \geq 1$  by the

existence of the edges of the triangle  $uvw$ ). By our assumption  $r$  is not a vertex of the triangle  $uvw$ . Thus  $r$  is contained by at most one of the sets  $X, Y, Z$ . Without loss of generality, suppose that  $r \notin X \cup Y$ . By (i) each of the subgraphs  $G[X], G[Y]$  contains an edge which is face contractible in  $G$ . These edges are distinct and avoid  $r$ . Thus  $G$  has two face contractible edges not containing  $r$ , as required.

Now suppose that  $uv$  is contractible but  $vw$  and  $uw$  are not contractible. Then we have maximal blockers  $X, Y$  for the edges  $vw, uw$ , respectively. As above, we must have  $X \cap Y = \{w\}$  by Lemma 4. Since  $r \neq w$ , we may assume, without loss of generality, that  $r \notin X$ . Then it follows from (i) that there is an edge  $f$  in  $G[X]$  which is face contractible in  $G$ . Thus we have two edges ( $uv$  and  $f$ ), not incident with  $r$ , which are face contractible in  $G$ .

**Case 2.** Each of the triangle faces of  $G$  contains  $r$ .

Consider a triangle face  $ruv$  of  $G$ . Then  $uv$  is face contractible, for otherwise (i) would imply that there is a face contractible edge in  $G[X]$ , (in particular, there is a triangle face of  $G$  which does not contain  $r$ ), a contradiction. Since  $G$  has at least two triangle faces by Lemma 2, it follows that  $G$  has at least two face contractible edges avoiding  $r$ . This proves the theorem.  $\square$

Theorem 4 implies that a plane Laman graph on at least four vertices has a face contractible edge. A plane Laman graph on three vertices is a triangle, and each of its edges is face contractible. Note that after the contraction of a face contractible edge the planar embedding of the resulting graph can be obtained by a simple local modification.

Since contracting an edge of a triangle face is the inverse operation of plane vertex splitting, the proof of the following theorem by induction is straightforward.

**Theorem 5.** *A graph is a plane Laman graph if and only if it can be obtained from an edge by plane vertex splitting operations.*

Theorem 4 implies that if  $G$  has at least four vertices then there is a face contractible edge avoiding any given triangle face. Thus any triangle face can be chosen as the starting configuration in Theorem 5.

We say that  $G = (V, E)$  is a *rigidity circuit* if  $G - e$  is Laman for every edge  $e$  of  $G$ . We call it a *Laman-plus-one graph* if  $G - e$  is Laman for some edge  $e$  of  $G$ . From the matroid viewpoint it is clear that Laman-plus-one graphs are those rigid graphs (i.e. spanning sets of the rigidity matroid) which contain a unique rigidity circuit. In particular, rigidity circuits are Laman-plus-one graphs.

By using similar techniques we can prove the following. The proof is omitted from this version.

**Theorem 6.** *A graph is a plane Laman-plus-one graph if and only if it can be obtained from a  $K_4$  by plane vertex splitting operations.*

**Remark** A natural question is whether a 3-connected plane Laman graph has a face contractible edge whose contraction preserves 3-connectivity as well (call

such an edge *strongly face contractible*). The answer is no: let  $G$  be a 3-connected plane Laman graph and let  $G'$  be obtained from  $G$  by inserting a new triangle face  $a'b'c'$  and adding the edges  $aa'$ ,  $bb'$ ,  $cc'$  (operation *triangle insertion*), for each of its triangle faces  $abc$ . Then  $G'$  is a 3-connected plane Laman graph with no strongly face contractible edges. It is an open question whether there exist good local reduction steps which could lead to an inductive construction in the 3-connected case, such that all intermediate graphs are also 3-connected.

## 4 Pseudo-Triangulations

Theorems 5 and 6 can be used to deduce a number of known results on pseudo-triangulations. One can give new proofs for Theorem 3 and for the following similar result (which was also proved in [6]): every planar Laman-plus-one graph can be embedded as a pointed-plus-one pseudo-triangulation. The proofs are omitted. They are similar to the proofs in [6] but the lengthy case analyses can be shortened, due to the fact that plane vertex splitting is more local.

Theorems 5 and 6 can also be used to prove the main results of [6] on *combinatorial pseudo-triangulations* of plane Laman and Laman-plus-one graphs. See [6] for more details on this combinatorial setting.

## 5 Planar Duals

Let  $G = (V, E)$  be a graph. We say that  $G$  is *co-Laman* if  $|E| = 2|V| - 1$  and  $i(X) \leq 2|X| - 2$  for all proper subsets  $X \subset V$ . Note that co-Laman graphs may contain multiple edges. Let  $M(G)$  denote the circuit matroid of  $G$  and let  $M^*(G)$  denote its dual.

**Theorem 7.** *Let  $G$  and  $H$  be a pair of planar graphs with  $M(G) \cong M^*(H)$ . Then  $G$  is Laman if and only if  $H$  is co-Laman.*

*Proof.* (Sketch) Simple matroid facts and Nash-Williams' characterisation of graphs which can be partitioned into two spanning trees imply that  $G$  is Laman  $\Leftrightarrow M(G/e)$  is the disjoint union of two bases (spanning trees) for all  $e \in E(G)$   $\Leftrightarrow M(H) - e$  is the disjoint union of two bases (spanning trees) for all  $e \in E(H)$   $\Leftrightarrow H$  is co-Laman.  $\square$

Recall the second Henneberg operation (edge splitting) from Section 1, which adds a new vertex and three new edges. If the third edge incident to the new vertex may also be connected to the endvertices of the removed edge (i.e. it may be parallel to one of the other new edges), we say that we perform a *weak edge splitting* operation. The *plane weak edge splitting* operation is the topological version of weak edge splitting (c.f. Lemma 1). It is not difficult to see that the dual of plane vertex splitting is plane weak edge splitting. Thus Theorem 5 and Theorem 7 imply the following inductive construction via planar duality.

**Theorem 8.** *A plane graph is co-Laman if and only if it can be obtained from a loop by plane weak edge splittings.*

## 6 Algorithms

In this section we describe the main ideas of an algorithm which can identify a face contractible edge in a plane Laman graph  $G$  in  $O(n^2)$  time, where  $n$  is the number of vertices of  $G$ . It uses a subroutine to test whether an edge  $e = uv$  of a triangle face is contractible (and to find its maximal blocker, if it is non-contractible), which is based on the following lemma. The maximal rigid subgraphs of a graph are called the *rigid components*, see [5,8] for more details.

**Lemma 10.** *Let  $G$  be a plane Laman graph and let  $e = uv$  be an edge of a triangle face  $uvw$  of  $G$ . Then either  $e$  is contractible, or the rigid component  $C$  of  $G - w$  containing  $e$  has at least three vertices. In the latter case the vertex set of  $C$  is the maximal blocker of  $e$  with respect to  $uvw$  in  $G$ .*

There exist algorithms for computing the rigid components of a graph in  $O(n^2)$  time, see e.g. [1,4,7].

The algorithm identifies a decreasing sequence  $V = X_0, X_1, \dots, X_t$  of subsets of  $V$  such that  $X_{i+1}$  is a blocker of some triangle edge of  $G[X_i]$  for  $0 \leq i \leq t-1$  and terminates with a contractible edge of  $G[X_t]$  which is also contractible in  $G$ . First it picks an arbitrary triangle face  $uvw$  of  $G$  and tests whether any of its edges is contractible (and computes the blockers, if they exist). If yes, it terminates with the desired contractible edge. If not, it picks the smallest blocker  $B$  found in this first iteration (which belongs to edge  $f = uv$ , say), puts  $X_1 = B$ , and continues the search for a proper edge in  $G[X_1]$ . To this end it picks a triangle in  $G[X_1]$  (which is different from the special face of  $G[X_1]$ , if  $X_1$  is not a nice blocker) and tests whether any of its edges which are different from  $f$  (and which are not on the special face of  $G[X_1]$ , if  $X_1$  is not a nice blocker) is contractible in  $G[X_1]$ . If yes, it terminates with that edge. Otherwise it picks the smallest blocker  $B'$  found in this iteration among those blockers which do not contain  $f$  (and which do not contain edges of the special face of  $G[X_1]$ , if  $X_1$  is not a nice blocker), puts  $X_2 = B'$ , and iterates. The following lemma, which can be proved by using Lemma 4, shows that there are always at least two blockers to choose from.

**Lemma 11.** (i) *Let  $f$  be an edge and let  $uvw$  be a triangle face whose edges (except  $f$ , if  $f$  is on the triangle) are non-contractible. Then at least two blockers of the edges of  $uvw$  do not contain  $f$ .*

(ii) *Let  $xyz$  and  $uvw$  be two distinct triangle faces such that the edges of  $uvw$  (except those edges which are common with  $xyz$ ) are non-contractible. Then at least two blockers of the edges of  $uvw$  do not contain edges of  $xyz$ .*

It follows from Lemma 8, Lemma 9, and Lemma 11 that the algorithm will eventually find a proper edge  $e$  in some subgraph  $G[X_t]$ , and that this edge will be face contractible in  $G$  as well. The fact that the algorithm picks the smallest blocker from at least two candidates (and that the blockers have exactly one vertex in common by Lemma 4) implies that the size of  $X_i$  will be essentially halved after every iteration. From this it is not difficult to see that the total running time is also  $O(n^2)$ .

**Theorem 9.** *A face contractible edge of a plane Laman graph can be found in  $O(n^2)$  time.*

Theorem 9 implies that an inductive construction via vertex splitting can be built in  $O(n^3)$  time. This matches the best time bound for finding a Henneberg construction, see e.g. [6].

## References

1. A. BERG AND T. JORDÁN, Algorithms for graph rigidity and scene analysis, Proc. 11th Annual European Symposium on Algorithms (ESA) 2003, (G. Di Battista, U. Zwick, eds) Springer Lecture Notes in Computer Science 2832, pp. 78-89, 2003.
2. A. BERG AND T. JORDÁN, A proof of Connelly's conjecture on 3-connected circuits of the rigidity matroid, *J. Combinatorial Theory, Ser. B.* Vol. 88, 77-97, 2003.
3. L. CHAVEZ, L. MOSHE, W. WHITELEY, Bases and circuits for 2-rigidity: constructions via tree coverings, preprint, Department of Mathematics and Statistics, York University, 2003.
4. H.N. GABOW AND H.H. WESTERMANN, Forests, frames and games: Algorithms for matroid sums and applications, *Algorithmica* 7, 465-497 (1992).
5. J. GRAVER, B. SERVATIUS, AND H. SERVATIUS, *Combinatorial Rigidity*, AMS Graduate Studies in Mathematics Vol. 2, 1993.
6. R. HAAS, D. ORDEN, G. ROTE, F. SANTOS, B. SERVATIUS, H. SERVATIUS, D. SOUVAINÉ, I. STREINU, W. WHITELEY, Planar minimally rigid graphs and pseudo-triangulations, Proc. 19th ACM Symposium on Computational Geometry, 2003, 154-163. Journal version to appear in *Computational Geometry, Theory and Applications* <http://www.arxiv.org/abs/math.CO/0307347>
7. B. HENDRICKSON, Conditions for unique graph realizations, *SIAM J. Comput.* **21** (1992), no. 1, 65-84.
8. B. JACKSON AND T. JORDÁN, Connected rigidity matroids and unique realizations of graphs, EGRES Technical Report 2002-12 ([www.cs.elte.hu/egres/](http://www.cs.elte.hu/egres/)), to appear in *J. Combin. Theory Ser. B.*
9. G. LAMAN, On graphs and rigidity of plane skeletal structures, *J. Engineering Math.* **4** (1970), 331-340.
10. L. LOVÁSZ AND Y. YEMINI, On generic rigidity in the plane, *SIAM J. Algebraic Discrete Methods* **3** (1982), no. 1, 91-98.
11. D. ORDEN, F. SANTOS, B. SERVATIUS, H. SERVATIUS, Combinatorial Pseudo Triangulations, preprint, arXiv:math.CO/0307370v1, 2003.
12. I. STREINU, A combinatorial approach to planar non-colliding robot arm motion planning, Proc. 41st FOCS, Redondo Beach, California, 2000, pp. 443-453.
13. I. STREINU, Combinatorial roadmaps in configuration spaces of simple planar polygons, Proceedings of the DIMACS Workshop on Algorithmic and Quantitative Aspects of Real Algebraic Geometry in Mathematics and Computer Science (Saugata Basu and Laureano Gonzalez-Vega, eds.) 2003, pp. 181-206.
14. T.S. TAY, W. WHITELEY, Generating isostatic frameworks, *Structural Topology* **11**, 1985, pp. 21-69.
15. W. WHITELEY, Vertex splitting in isostatic Frameworks, *Structural Topology* **16**, 23-30, 1991.
16. W. WHITELEY, Some matroids from discrete applied geometry. Matroid theory (Seattle, WA, 1995), 171-311, Contemp. Math., 197, Amer. Math. Soc., Providence, RI, 1996.



# Faster Fixed-Parameter Tractable Algorithms for Matching and Packing Problems<sup>\*</sup>

Michael R. Fellows<sup>1</sup>, C. Knauer<sup>2</sup>, N. Nishimura<sup>3, \*\*</sup>, P. Ragde<sup>3</sup>, F. Rosamond<sup>1</sup>,  
U. Stege<sup>4</sup>, Dimitrios M. Thilikos<sup>5</sup>, and S. Whitesides<sup>6</sup>

<sup>1</sup> School of Electrical Engineering and Computer Science, University of Newcastle,  
Australia

<sup>2</sup> Institute of Computer Science, Freie Universität Berlin, Germany

<sup>3</sup> School of Computer Science, University of Waterloo, Canada [nishi@uwaterloo.ca](mailto:nishi@uwaterloo.ca)

<sup>4</sup> Department of Computer Science, University of Victoria, Canada

<sup>5</sup> Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de  
Catalunya, Spain. <sup>¶</sup>

<sup>6</sup> School of Computer Science, McGill University, Canada

**Abstract.** We obtain faster algorithms for problems such as  $r$ -dimensional matching,  $r$ -set packing, graph packing, and graph edge packing when the size  $k$  of the solution is considered a parameter. We first establish a general framework for finding and exploiting small problem kernels (of size polynomial in  $k$ ). Previously such a kernel was known only for triangle packing. This technique lets us combine, in a new and sophisticated way, Alon, Yuster and Zwick's color-coding technique with dynamic programming on the structure of the kernel to obtain faster fixed-parameter algorithms for these problems. Our algorithms run in time  $O(n + 2^{O(k)})$ , an improvement over previous algorithms for some of these problems running in time  $O(n + k^{O(k)})$ . The flexibility of our approach allows tuning of algorithms to obtain smaller constants in the exponent.

## 1 Introduction

In this paper we demonstrate a general method for solving parameterized packing and matching problems by first finding a problem kernel, and then showing how color-coding (the use of nice families of hash functions to color a substructure with distinct colors) and dynamic programming on subsets of colors can be used to create fixed-parameter algorithms. The problems we consider include parameterized versions of  $r$ -dimensional matching (a generalization of 3-dimensional matching, from Karp's original list of NP-complete problems [Kar72]),  $r$ -set

---

<sup>\*</sup> Research initiated at the International Workshop on Fixed Parameter Tractability in Computational Geometry and Games, Bellairs Research Institute of McGill University, Holetown, Barbados, Feb. 7-13, 2004, organized by S. Whitesides.

<sup>\*\*</sup> Contact author.

<sup>¶</sup> Supported by the EU within the 6th Framework Programme under contract 001907 (DELIS) and by the Spanish CICYT project TIC-2002-04498-C05-03 (TRACER).



packing (on Karp’s list in its unrestricted form, and  $W[1]$ -complete [ADP80] in its unrestricted parameterized form, hence unlikely to be fixed-parameter tractable), graph packing (a generalization of subgraph isomorphism known to be NP-complete [Coo71]), and graph edge packing (shown to be NP-complete by Holyer [Hol81]). Further generalizations are considered at the end of the paper.

Previous efforts in finding fixed-parameter algorithms for these problems (asking if a given input has a solution of size at least  $k$ ) have resulted in running times involving a factor of  $k^{O(k)}$ , namely  $O((5.7k)^k \cdot n)$  for 3-dimensional matching [CFJK01] and 3-set packing [JZC04]. Earlier work did not use the technique of kernelization (finding a subproblem of size  $f(k)$  within which any solution of size  $k$  must lie) except for work on the problem of packing triangles [FHR<sup>+</sup>04]; that result uses the technique of crown decomposition [Fel03, CFJ04], a technique that is not used in this paper (concurrent with this paper are further uses of this technique for packing triangles [MPS04] and stars [PS04]). Using our techniques, we improve all these results by replacing the  $k^{O(k)}$  factor with a factor of  $2^{O(k)}$ . Kernelization also allows us to make the dependence on  $k$  additive, that is, we can achieve a running time of  $O(n + 2^{O(k)})$  (the hidden constant in the exponent is linearly dependent on  $r$ ).

The techniques we introduce make use of the fact that our goal is to obtain at least  $k$  disjoint objects, each represented as a tuple of values. We first form a maximal set of disjoint objects and then use this set both as a way to bound the number of values outside the set (forming the kernel) and as a tool in the color-coding dynamic programming (forming the algorithm). We are able to reduce the size of the constants inherent in the choice of hash functions in the original formulation of color-coding [AYZ95] by applying the coloring to our kernels only, and using smaller families of hash functions with weaker properties. Although both color-coding and dynamic programming across subsets appear in other work [Mar04, Woe03, CFJ04], our technique breaks new ground by refining these ideas in their joint use.

After introducing notation common to all the problems in Section 2, we first present the kernelization algorithm for  $r$ -DIMENSIONAL MATCHING (Section 3), followed by the fixed-parameter algorithm in Section 4. Next, we consider the modifications necessary to solve the other problems in Section 5. The paper concludes with a discussion of tuning the hash functions (Section 6) and further generalization of our techniques (Section 7).

## 2 Definitions

Each of the problems considered in this paper requires the determination of  $k$  or more disjoint structures within the given input: in  $r$ -dimensional matching, we find  $k$  points so that no coordinates are repeated; in  $r$ -set packing, we find  $k$  disjoint sets; in graph packing, we find  $k$  vertex-disjoint subgraphs isomorphic to input  $H$ ; and in graph edge packing, we find  $k$  edge-disjoint subgraphs isomorphic to input  $H$ . To unify the approach taken to the problems, we view each structure as an  $r$ -tuple (where in the last two cases  $r = |V(H)|$  and  $r = |E(H)|$ ),

respectively); the problems differ in the ways the  $r$ -tuples must be disjoint and, in the case of graph packing, on additional constraints put on the structures.

To define the  $r$ -tuples, we consider a collection  $A_1, \dots, A_r$  of pair-wise disjoint sets and define  $\mathcal{A}$  to be  $A_1 \times \dots \times A_r$ . We call the elements of each  $A_i$  *values*; given an  $r$ -tuple  $T \in \mathcal{A}$ , we denote as  $\text{val}(T)$  the set of values of  $T$ , and for any  $\mathcal{S} \subseteq \mathcal{A}$ , we define  $\text{val}(\mathcal{S}) = \bigcup_{T \in \mathcal{S}} \text{val}(T)$ . Depending on the problem, we will either wish to ensure that tuples chosen have disjoint sets of values or, for  $r$ -dimensional matching, that tuples “disagree” at a particular coordinate.

We introduce further notation to handle the discussion of  $r$ -tuples. Given a tuple  $T \in \mathcal{A}$ , the  $i$ th coordinate of  $T$ , denoted  $T(i)$ , is the value of  $T$  from set  $A_i$ . Two  $r$ -tuples  $T$  and  $T'$  are *linked* if they agree in some coordinate  $i$ , that is, when there exists some  $i$  in  $\{1, \dots, r\}$  such that  $T(i) = T'(i)$ . We denote this fact as  $T \sim T'$ . If  $T \sim T'$  does not hold then we say that  $T$  and  $T'$  are *independent*, denoted  $T \not\sim T'$ .

We can now define our example problem,  $r$ -DIMENSIONAL MATCHING, in which the input is a set of  $r$ -tuples and the goal is to find at least  $k$  independent tuples. The precise descriptions of the other problems will be deferred to Section 5.

$r$ -DIMENSIONAL MATCHING

*Instance:* A set  $\mathcal{S} \subseteq \mathcal{A} = A_1 \times \dots \times A_r$  for some collection  $A_1, \dots, A_r$  of pair-wise disjoint sets.

*Parameter:* A non-negative integer  $k$ .

*Question:* Is there a matching  $\mathcal{P}$  for  $\mathcal{S}$  of size at least  $k$ , that is, is there a subset  $\mathcal{P}$  of  $\mathcal{S}$  where for any  $T, T' \in \mathcal{P}$ ,  $T \not\sim T'$  and  $|\mathcal{P}| \geq k$ ?

In forming a kernel, we will be able to reduce the number of tuples that contain a particular value or set of values. To describe a set of tuples in which some values are specified and others may range freely, we will need an augmented version  $A_i^*$  of each  $A_i$  so that  $A_i^* = A_i \cup \{*\}$ ,  $i \in \{1 \dots, r\}$ ; the stars will be used to denote positions at which values are unrestricted. We will refer to special tuples, *patterns*, drawn from the set  $\mathcal{A}^* = A_1^* \times \dots \times A_r^*$ . Associated with each pattern will be a corresponding set of tuples. More formally, given  $R \in \mathcal{A}^*$ , we set  $\mathcal{A}[R] = A'_1 \times \dots \times A'_r$  where

$$A'_i = \begin{cases} A_i & \text{if } R(i) = * \\ \{R(i)\} & \text{otherwise} \end{cases}$$

As we will typically wish to consider the subset of  $\mathcal{S}$  extracted using the pattern, we further define, for any  $\mathcal{S} \subseteq \mathcal{A}$ , the set  $\mathcal{S}[R] = \mathcal{A}[R] \cap \mathcal{S}$ . We finally define  $\text{free}(R)$  as the number of  $*$ 's in  $R$ , namely the number of unrestricted positions in the pattern;  $\text{free}(R)$  will be called the *freedom of  $R$* .

### 3 A Kernel for $r$ -DIMENSIONAL MATCHING

Recall that a kernel is a subproblem of size depending only on  $k$  within which any solution of size  $k$  must lie. The idea behind our kernelization technique is

that if a large number of tuples match a particular pattern, we can remove some of them (this is known as a *reduction rule*). For each pattern  $R$ , the threshold size for the set of retained tuples is a function  $f(\text{free}(R), k)$ , which we define later.

The following gives a general family of reduction rules for the  $r$ -DIMENSIONAL MATCHING problem with input  $\mathcal{S}_I$  and parameter  $k$ . For each application of the function **Reduce**, if the size of the subset  $\mathcal{S}_I[R]$  of  $\mathcal{S}$  extracted using the pattern  $R$  is greater than the threshold, then  $|\mathcal{S}|$  is reduced by removing enough elements of  $\mathcal{S}_I[R]$  to match the size of the function  $f$ .

Function **Reduce**( $\mathcal{S}_I, R$ ) where  $R \in \mathcal{A}^*$  and  $1 \leq \text{free}(R) \leq r - 1$ .  
*Input:* A set  $\mathcal{S}_I \subseteq \mathcal{A}$  and a pattern  $R \in \mathcal{A}^*$ .  
*Output:* A set  $\mathcal{S}_O \subseteq \mathcal{S}_I$ .  
 $\mathcal{S}_O \leftarrow \mathcal{S}_I$ .  
**If**  $|\mathcal{S}_I[R]| > f(\text{free}(R), k)$   
 then remove all but  $f(\text{free}(R), k)$  tuples of  $\mathcal{S}_I[R]$  from  $\mathcal{S}_O$ .  
**output**  $\mathcal{S}_O$ .

To form the kernel, we apply the function **Reduce** on all patterns  $R \in \mathcal{A}^*$  in order of increasing freedom, as in the following routine.

Function **Fully-Reduce**( $\mathcal{S}_I$ )  
*Input:* A set  $\mathcal{S}_I \subseteq \mathcal{A}$ .  
*Output:* A set  $\mathcal{S}_O \subseteq \mathcal{S}_I$ .  
 $\mathcal{S}_O \leftarrow \mathcal{S}_I$ .  
**For**  $i = 1, \dots, r - 1$  **do**  
     **for all**  $R \in \mathcal{A}^*$  where  $\text{free}(R) = i$ ,  $\mathcal{S}_O \leftarrow \text{Reduce}(\mathcal{S}_O, R)$ .  
**output**  $\mathcal{S}_O$ .

The next three lemmas prove that the above function computes a kernel; the two lemmas after that show how a slight modification can be used to bound the size of that kernel.

We say that a set  $\mathcal{S} \subseteq \mathcal{A}$  is  $(\ell, k)$ -*reduced* if for any  $R \in \mathcal{A}^*$  such that  $\text{free}(R) = \ell$ ,  $\text{Reduce}(\mathcal{S}, R) = \mathcal{S}$  (where  $1 \leq \ell \leq r - 1$ ). For notational convenience we define any set  $\mathcal{S} \subseteq \mathcal{A}$  to be 0-reduced; we can assume the input has no repeated tuples. As a direct consequence of the definition, if a set  $\mathcal{S}$  is  $(\ell, k)$ -reduced, then for each pattern  $R$  such that  $\text{free}(R) = \ell$ ,  $|\mathcal{S}[R]| \leq f(\ell, k)$ .

We now show that our reduction function does not change a yes-instance of  $r$ -DIMENSIONAL MATCHING into a no-instance, nor vice versa. Given a set  $\mathcal{S} \subseteq \mathcal{A}$  and a non-negative integer  $k$ , we denote as  $\mu(\mathcal{S}, k)$  the set of all matchings for  $\mathcal{S}$  of size at least  $k$ . If  $\mu(\mathcal{S}, k)$  is non-empty,  $\mathcal{S}$  is a yes-instance. The following lemma shows that upon applying the reduction rule to an  $(\ell - 1, k)$ -reduced set using a pattern  $R$  of freedom  $\ell$ , yes-instances will remain yes-instances. The proof requires the precise definition of  $f(\ell, k)$ , which is that  $f(0, k) = 1$  and  $f(\ell, k) = \ell \cdot (k - 1) \cdot f(\ell - 1, k) + 1$  for all  $\ell > 0$ . It is easy to see that  $f(\ell, k) \leq \ell! k^\ell$ .

**Lemma 1.** *For any  $\ell, 1 \leq \ell \leq r - 1$ , the following holds: If  $\mathcal{S} \subseteq \mathcal{A}$ ,  $\mathcal{S}$  is  $(\ell - 1, k)$ -reduced, and  $\mathcal{S}' = \text{Reduce}(\mathcal{S}, R)$  for some  $R$  such that  $\text{free}(R) = \ell$ , then  $\mu(\mathcal{S}, k) \neq \emptyset$  if and only if  $\mu(\mathcal{S}', k) \neq \emptyset$ .*

**Proof.** Since  $\mathcal{S}' \subseteq \mathcal{S}$ ,  $\mu(\mathcal{S}', k) \neq \emptyset$  implies  $\mu(\mathcal{S}, k) \neq \emptyset$ . Supposing now that  $\mu(\mathcal{S}, k) \neq \emptyset$ , we choose  $\mathcal{P} \in \mu(\mathcal{S}, k)$  where  $|\mathcal{P}| = k$  and prove that  $\mu(\mathcal{S}', k) \neq \emptyset$ . In essence, we need to show that either  $\mathcal{P} \in \mu(\mathcal{S}', k)$  or that we can form a matching  $\mathcal{P}' \in \mu(\mathcal{S}', k)$  using some of the tuples in  $\mathcal{P}$ .

Denote by  $\hat{\mathcal{S}}$  the set  $\mathcal{S}'[R]$ , that is, the tuples retained when **Reduce** is applied to  $\mathcal{S}$  using pattern  $R$  to form  $\mathcal{S}'$ . Clearly if either  $\mathcal{P}$  contained no tuple in  $\mathcal{S}[R]$  or the single tuple  $T \in \mathcal{P} \cap \mathcal{S}[R]$  was selected to be in  $\hat{\mathcal{S}}$ , then  $\mathcal{P} \in \mu(\mathcal{S}', k)$ , completing the proof in these two cases.

In the case where  $T \in \mathcal{P}$  and  $T \notin \hat{\mathcal{S}}$ , we show that there is a tuple  $T' \in \hat{\mathcal{S}}$  that can replace  $T$  to form a new matching, formalized in the claim below.

*Claim:* There exists a  $T' \in \hat{\mathcal{S}}$  that is independent from each  $\tilde{T} \in \mathcal{P} - \{T\}$ .

*Proof:* We first show that for any  $i \in \{j \mid R(j) = *\}$  and any  $\tilde{T} \in \mathcal{P} - \{T\}$  there exist at most  $f(\ell - 1, k)$   $r$ -tuples in  $\hat{\mathcal{S}}$  that agree with  $\tilde{T}$  at position  $i$ . The set of  $r$ -tuples in  $\mathcal{S}[R]$  that agree with  $\tilde{T}$  at position  $i$  is exactly the set of tuples in  $\mathcal{S}[R']$  where  $R'$  is obtained from  $R$  by replacing the  $*$  in position  $i$  by  $\tilde{T}(i)$ . We use the size of this set to bound the size of the set of tuples in  $\hat{\mathcal{S}}$  that agree with  $\tilde{T}$  at position  $i$ . As  $\mathcal{S}$  is  $(\ell - 1, k)$ -reduced and  $\text{free}(R') = \ell - 1$ , we know that  $|\mathcal{S}[R']| \leq f(\ell - 1, k)$ . Since in the special case where  $\ell = 1$ , we directly have  $|\mathcal{S}[R']| \leq 1 = f(0, k)$ , the bound of  $f(\ell - 1, k)$  holds for any  $\tilde{T}$  and any  $i$ .

To complete the proof of the claim, we determine the number of elements of  $\hat{\mathcal{S}}$  that can be linked with any  $\tilde{T}$  and show that at least one member of  $\hat{\mathcal{S}}$  is not linked with any in the set  $\mathcal{P} - \{T\}$ . Using the statement proved above, as  $|\{j \mid R(j) = *\}| = \ell$  and  $|\mathcal{P} - \{T\}| = k - 1$ , at most  $\ell \cdot (k - 1) \cdot f(\ell - 1, k) = f(\ell, k) - 1$  elements of  $\hat{\mathcal{S}}$  will be linked with elements of  $\mathcal{P} - \{T\}$ . Since  $|\hat{\mathcal{S}}| = f(\ell, k)$ , this implies the existence of some  $T' \in \hat{\mathcal{S}}$  with the required property.

The claim above implies that  $\mathcal{P}' = (\mathcal{P} - \{T\}) \cup \{T'\}$  is a matching of  $\mathcal{S}$  of size  $k$ . As  $T' \in \hat{\mathcal{S}}$ ,  $\mathcal{P}'$  is also a matching of  $\mathcal{S}'$ , and thus  $\mu(\mathcal{S}', k) \neq \emptyset$ .  $\square$

Lemma 2 can be proved by induction on the number of iterations; proofs for this and subsequent lemmas are omitted due to space limitations.

**Lemma 2.** *If  $\mathcal{S} \subseteq \mathcal{A}$  and  $\mathcal{S}'$  is the output of routine **Fully-Reduce**( $\mathcal{S}$ ), then (a)  $\mathcal{S}'$  is an  $(r - 1, k)$ -reduced set and (b)  $\mu(\mathcal{S}, k) \neq \emptyset$  if and only if  $\mu(\mathcal{S}', k) \neq \emptyset$ .*

We now use a maximal matching in conjunction with the function **Fully-Reduce** defined above to bound the size of the kernel. The following lemma is a direct consequence of the definition of an  $(r - 1, k)$ -reduced set.

**Lemma 3.** *For any  $(r - 1, k)$ -reduced set  $\mathcal{S} \subseteq \mathcal{A}$ , any value  $x \in \text{val}(\mathcal{A})$  is contained in at most  $f(r - 1, k)$   $r$ -tuples of  $\mathcal{S}$ .*

We now observe that if we have a maximal matching in a set  $\mathcal{S}$  of  $r$ -tuples, we can bound the size of the set as a function of  $r$ , the size of the matching, and  $f(r - 1, k)$ , as each  $r$ -tuple in the set must be linked with at least one  $r$ -tuple in the matching, and the number of such links is bounded by  $f(r - 1, k)$  per value.

**Lemma 4.** *If  $\mathcal{S} \subseteq \mathcal{A}$  is an  $(r-1, k)$ -reduced set containing a maximal matching  $\mathcal{M}$  of size at most  $m$ , then  $\mathcal{S}$  contains no more than  $r \cdot m \cdot f(r-1, k)$   $r$ -tuples.*

Lemma 4 suggests a kernel for the  $r$ -DIMENSIONAL MATCHING problem in the function that follows; Lemma 5 shows the correctness of the procedure and size of the kernel.

Function **Kernel-Construct**( $\mathcal{S}$ )

*Input:* A set  $\mathcal{S} \subseteq \mathcal{A}$ .

*Output:* A set  $\mathcal{K} \subseteq \mathcal{S}$ .

$\mathcal{S}' \leftarrow \text{Fully-Reduce}(\mathcal{S})$ .

**Find** a maximal matching  $\mathcal{M}$  of  $\mathcal{S}'$ .

**If**  $|\mathcal{M}| \geq k$  then **output** any subset  $\mathcal{K}$  of  $\mathcal{M}$  of size  $k$  and **stop**;  
otherwise **output**  $\mathcal{K} \leftarrow \mathcal{S}'$ .

**Lemma 5.** *If  $\mathcal{S} \subseteq \mathcal{A}$  and  $\mathcal{K}$  is the output of the function **Kernel-Construct**( $\mathcal{S}$ ), then (a)  $|\mathcal{K}| \in O(k^r)$  and (b)  $\mu(\mathcal{S}, k) \neq \emptyset$  if and only if  $\mu(\mathcal{K}, k) \neq \emptyset$ .*

Note that function **Kernel-Construct** can be computed in time  $O(n)$  for fixed  $r$ , simply by looking at each tuple in turn and incrementing the counter associated with each of the  $2^r - 1$  patterns derived from it, deleting the tuple if any such counter overflows its threshold.

## 4 An FPT Algorithm for $r$ -DIMENSIONAL MATCHING

While it suffices to restrict attention to the kernel  $\mathcal{K}$  when seeking a matching of size  $k$ , exhaustive search of  $\mathcal{K}$  does not lead to a fast algorithm. Hence we propose a novel alternative, which combines the colour coding technique of Alon et al. [AYZ95] with the use of a maximal matching  $\mathcal{M} \subseteq \mathcal{K}$ .

In its original form, the colour-coding technique of Alon et al. makes use of a family of hash functions  $\mathcal{F} = \{f : U \rightarrow X\}$  with the property that for any  $S \subseteq U$  with  $|S| \leq |X|$ , there is an  $f \in \mathcal{F}$  that is 1-1 on  $S$ . The original idea, in the context of our problem, would be to look for a matching of size  $k$  whose values receive distinct colours under some  $f \in \mathcal{F}$ . In our case, the universe  $U$  is the set of values appearing in tuples in the kernel of the problem, which has size  $O(k^r)$ , and the set of colours  $X$  has size  $rk$ . The family of hash functions  $\mathcal{F}$  used by Alon et al. is of size  $2^{O(rk)}$  in our case. In Section 6 we will consider modifications towards improving this approach.

To modify colour-coding for our purposes, we first compute a maximal matching  $\mathcal{M}$  in the kernel. By the maximality of  $\mathcal{M}$ , each  $r$ -tuple in a matching  $\mathcal{P}$  of size  $k$  in the kernel must contain a value in  $\mathcal{M}$ . Hence there may be at most  $(r-1)k$  values of  $\mathcal{P}$  that do not belong to  $\mathcal{M}$ . We will seek a proper colouring of these values. Thus we choose a universe  $U = \text{val}(\mathcal{K}) - \text{val}(\mathcal{M})$ , and we set  $|X| = (r-1)k$ .

Our dynamic programming problem space has four dimensions. Two of them are the choice of hash function  $\phi$ , and a subset  $W$  of values of  $\mathcal{M}$  that might be

used by  $\mathcal{P}$ . For each choice of these, there are two more dimensions associated with a possible subset  $\mathcal{P}'$  of  $\mathcal{P}$ : the set of values  $Z$  in  $W$  it uses, and the set of colours  $C$  that  $\phi$  assigns its values not in  $W$ . More formally, for  $\mathcal{M}$  a maximal matching of  $\mathcal{S} \subseteq \mathcal{A}$ , then for any  $W \subseteq \text{val}(\mathcal{M})$ ,  $\phi \in \mathcal{F}$ ,  $Z \subseteq W$ , and  $C \subseteq X$  such that  $|Z| + |C| \leq r \cdot |\mathcal{P}|$  and  $|Z| + |C| \equiv 0 \pmod{r}$  we define

$$B_{\phi, W}(Z, C) = \begin{cases} 1 & \text{if there exists a matching } \mathcal{P}' \subseteq \mathcal{S} \text{ where } |\mathcal{P}'| = \frac{|Z|+|C|}{r}, \\ & \text{val}(\mathcal{P}') \cap \text{val}(\mathcal{M}) = Z, \text{ and } \phi(\text{val}(\mathcal{P}') - Z) = C, \\ 0 & \text{otherwise} \end{cases}$$

where for convenience we use the notation  $\phi(S)$  for a set  $S$  to be  $\cup_{v \in S} \phi(v)$ . In order to solve the problem, our goal is then to use dynamic programming to determine  $B_{\phi, W}(Z, C)$  for each  $W$ , for each  $\phi$  in the family  $\mathcal{F}$ , and for each  $Z$  and  $C$  such that  $|Z| + |C| \leq rk$ .

To count the number of dynamic programming problems thus defined, we note that there are at most  $2^{(r-1)k}$  choices for  $W$ ,  $C$ , and  $Z$ , and  $2^{O(rk)}$  choices of  $\phi \in \mathcal{F}$ . Thus there are  $2^{O(rk)}$  problems in total.

To formulate the recurrence for our dynamic programming problem, we note that  $B_{\phi, W}(Z, C) = 1$  for  $|Z| + |C| = 0$ , and observe that  $B_{\phi, W}(Z, C) = 1$ , for  $|Z| + |C| > 0$ , holds precisely when there exists an  $r$ -tuple formed of a subset  $Z'$  of  $Z$  and a subset  $C'$  of  $C$  such that there is a matching of size one smaller using  $Z - Z'$  and  $C - C'$ . For the ease of exposition, we define the function  $P_{\phi} : 2^W \times 2^X \rightarrow \{0, 1\}$  (computable in  $O(k^r)$  time) as

$$P_{\phi}(Z', C') = \begin{cases} 1 & \text{if there exists an } r\text{-tuple } T \in \mathcal{S} \text{ where } Z' \subseteq \text{val}(T) \\ & \text{and } \phi(\text{val}(T) - Z') = C', \\ 0 & \text{otherwise} \end{cases}$$

Observing that each  $r$ -tuple must contain at least one element in  $Z$ , we can then calculate  $B_{\phi, W}(Z, C)$  by dynamic programming as follows:

$$B_{\phi, W}(Z, C) = \begin{cases} 1 & \text{if there exist } Z' \subseteq Z, C' \subseteq C \text{ with } |Z'| \geq 1, |Z'| + |C'| = r, \\ & P_{\phi}(Z', C') = 1 \text{ and } B_{\phi, W}(Z - Z', C - C') = 1 \\ 0 & \text{otherwise} \end{cases}$$

One table entry can be computed in  $O(k^r)$  time. The number of choices for  $Z'$  and  $C'$  can be bounded by  $\binom{|Z|+|C|}{r} \leq \binom{rk}{r} = O(k^r)$ . The algorithm below is a summary of the above description, where  $\mathcal{F}$  is a set of hash functions from  $\text{val}(\mathcal{S}) - \text{val}(\mathcal{M})$  to  $X$ .

In merging the following routine with the kernel construction, the maximal matching needs to be found only once. To find a maximal matching in the kernel, we use a greedy algorithm running in time  $O(k^r)$ . The running time is dominated by kernel construction (taking time  $O(n)$ ) plus dynamic programming (taking time  $2^{O(rk)}$ ). This yields the following theorem.

**Theorem 1.** *The problem  $r$ -DIMENSIONAL MATCHING can be solved in time  $O(n + 2^{O(rk)})$ .*

**Routine Color-Coding-Matching( $\mathcal{S}$ )***Input:* A set  $\mathcal{S} \subseteq \mathcal{A}$ .*Output:* A member of the set {Yes, No}.**Find** a maximal matching  $\mathcal{M}$  of  $\mathcal{S}$ .**If**  $|\mathcal{M}| \geq k$  then **output** “Yes” and **stop**.**set**  $V = \text{val}(\mathcal{S}) - \text{val}(\mathcal{M})$ .**For** each  $W \subseteq \text{val}(\mathcal{M})$ , **do**    **for** each coloring  $\phi : V \rightarrow X$  where  $\phi \in \mathcal{F}$         **for** each  $Z \subseteq W$  by increasing  $|Z|$             **for** each  $C \subseteq X$  by increasing  $|C|$ , where  $|Z| + |C| \equiv 0 \pmod r$   
                compute  $B_{\phi, W}(Z, C)$ .                **if**  $B_{\phi, W}(Z, C) = 1$  and  $|Z| + |C| = rk$ , **output** “Yes” and **stop**.**Output** “No”.

## 5 Kernels and FPT Algorithms for the Other Problems

We can now define other problems addressable by our technique. To avoid introducing new notation, each of them will be defined in terms of sets of tuples, even though order within a tuple is not important in some cases. For  $r$ -SET PACKING, the input  $r$ -tuples are subsets of elements from a base set  $A$ , each of size at most  $r$ , and the goal is to find at least  $k$   $r$ -tuples, none of which share elements.

 *$r$ -SET PACKING**Instance:* A collection  $\mathcal{C}$  of sets drawn from a set  $A$ , each of size of at most  $r$ .*Parameter:* A non-negative integer  $k$ .*Question:* Does  $\mathcal{C}$  contain at least  $k$  mutually disjoint sets, that is, for  $\mathcal{S} \subseteq \mathcal{A} = A^r$ , is there a subset  $\mathcal{P}$  of  $\mathcal{S}$  where for any  $T, T' \in \mathcal{P}$ ,  $\text{val}(T) \cap \text{val}(T') = \emptyset$  and  $|\mathcal{P}| \geq k$ ?

In order to define the graph problems, we define  $G[S]$  to be the subgraph of  $G$  induced on the vertex set  $S \subseteq V(G)$ , namely the graph  $G' = (V(G'), E(G'))$ , where  $V(G') = S$  and  $E(G') = \{(u, v) \mid u, v \in S, (u, v) \in E(G)\}$ . Moreover, we use  $H \subseteq G$  to denote that  $H$  is a (not necessarily induced) subgraph of  $G$ .

**GRAPH PACKING***Instance:* Two graphs  $G = (V(G), E(G))$  and  $H = (V(H), E(H))$ .*Parameter:* A non-negative integer  $k$ .*Question:* Does  $G$  contain at least  $k$  vertex-disjoint subgraphs each isomorphic to  $H$ , that is, for  $\mathcal{S} \subseteq \mathcal{A} = V(G)^{|V(H)|}$ , is there a subset  $\mathcal{P}$  of  $\mathcal{S}$  where for any  $T, T' \in \mathcal{P}$ ,  $\text{val}(T) \cap \text{val}(T') = \emptyset$ , there is a subgraph  $G' \subseteq G[\text{val}(T)]$  that is isomorphic to  $H$  for any  $T \in \mathcal{P}$ , and  $|\mathcal{P}| \geq k$ ?**GRAPH EDGE-PACKING***Instance:* Two graphs  $G = (V(G), E(G))$  and  $H = (V(H), E(H))$  such that  $H$  has no isolated vertices.*Parameter:* A non-negative integer  $k$ .*Question:* Does  $G$  contain at least  $k$  edge-disjoint subgraphs each isomorphic to  $H$ , that is, for  $\mathcal{S} \subseteq \mathcal{A} = E(G)^{|E(H)|}$ , is there a subset  $\mathcal{P}$  of  $\mathcal{S}$  where for any



$T, T' \in \mathcal{P}$ ,  $\text{val}(T) \cap \text{val}(T') = \emptyset$ , there is a subgraph  $G'$  that is isomorphic to  $H$  such that each edge of  $G'$  is in  $T$  for any  $T \in \mathcal{P}$ , and  $|\mathcal{P}| \geq k$ ?

To obtain kernels for the above problems, we can adapt the ideas developed for  $r$ -DIMENSIONAL MATCHING. As in that case, we first apply a reduction rule that limits the number of tuples containing a particular value or set of values, and then use a maximal set of disjoint objects to bound the size of the kernel. In the remainder of this section we detail the differences among the solutions for the various problems.

For each of these problems, the tuples in question are drawn from the same base set ( $A$ ,  $V(G)$ , or  $E(G)$ ) respectively, resulting in the tuples being sets of size at most  $r$ , as position within a tuple is no longer important. Since in the last two problems there is an additional constraint that the set of vertices or edges forms a graph isomorphic to  $H$ , the sets are forced to be of size exactly  $r$ ; for  $r$ -SET PACKING there is no such constraint, allowing smaller sets.

For all three problems, since the  $A_i$ 's are no longer disjoint, the potential for conflicts increases. As a consequence, we define a new function  $g$  for use in the function Reduce:  $g(0, k) = 1$  and  $g(\ell, k) = r \cdot \ell \cdot (k - 1) \cdot g(\ell - 1, k) + 1$  for all  $\ell > 0$ , for  $r = |V(H)|$  and  $r = |E(H)|$  in the latter two problems. By replacing  $f$  by  $g$  in the definition of the function Reduce, we obtain a new function SetReduce and the notion of a set being  $(\ell, k)$ -set-reduced, and by replacing Reduce by SetReduce in the function Fully-Reduce, we can obtain a new function Fully-SetReduce. Instead of finding a maximal matching, we find a maximal set of disjoint sets or a maximal set of disjoint sets of vertices or edges yielding subgraphs isomorphic to  $H$  in the function Kernel-Construct. It then remains to prove analogues of Lemmas 1 through 5; sketches of the changes are given below.

Each of the lemmas can be modified by replacing  $f$  by  $g$ , Reduce by SetReduce, and Fully-Reduce by Fully-SetReduce, with the analogue of Lemma 5 yielding a kernel of size  $O(k^r)$ . The need for  $g$  instead of  $f$  is evident in the proof of the claim in the analogue of Lemma 1. Here we count the number of  $r$ -tuples  $\hat{T}$  in  $\hat{S}$  such that  $\text{val}(\hat{T}) \cap \text{val}(\tilde{T}) \neq \emptyset$ . Since the  $r$ -tuples are in fact sets, positions have no meaning, and hence the number of such  $r$ -tuples will be the number of positions per tuple (that is,  $r$ ) multiplied by the number of free positions ( $\ell$  in any  $R'$  formed by fixing one more index) multiplied by the number of tuples in  $\mathcal{P} - \{\mathcal{T}\}$  (that is,  $k - 1$ ) multiplied by  $g(\ell - 1, k)$ . In total this gives us  $r \cdot \ell \cdot (k - 1) \cdot g(\ell - 1, k) = g(\ell, k) - 1$  elements of  $\hat{S}$  with nonempty intersection with values in elements of  $\mathcal{P} - \{\mathcal{T}\}$ , as needed to show that there is a  $T'$  with the required property. We then have the following lemma, analogous to Lemma 5:

**Lemma 6.** *For the problems  $r$ -SET PACKING, GRAPH PACKING, and GRAPH EDGE-PACKING, if  $\mathcal{S} \subseteq \mathcal{A}$  and  $\mathcal{K}$  is the output of the function Kernel-Construct( $\mathcal{S}$ ), then (a)  $|\mathcal{K}| \in O(r^r k^r)$  and (b)  $\mathcal{S}$  has a set of at least  $k$  objects (disjoint sets, vertex-disjoint subgraphs isomorphic to  $H$ , or edge-disjoint subgraphs isomorphic to  $H$ , respectively) if and only if  $\mathcal{K}$  has a set of at least  $k$  such objects.*

To obtain algorithms for these problems, we adapt the ideas developed for  $r$ -DIMENSIONAL MATCHING; as in that case we form a kernel, find a maximal set



of disjoint sets or graphs, and then use dynamic programming and color-coding to find a solution of size  $k$ , if one exists.

Our algorithms differ from that for  $r$ -DIMENSIONAL MATCHING only in the definitions of  $B_{\phi,W}(Z, C)$  and  $P_{\phi}$ . In particular, the conditions for  $B_{\phi,W}(Z, C)$  to be 1 are as follows: there exists a set  $\mathcal{P} \subseteq \mathcal{S}$  of disjoint sets where  $|\mathcal{P}| \geq \frac{|Z|+|C|}{r}$ ,  $\text{val}(\mathcal{P}) \cap \text{val}(\mathcal{M}) = Z$ , and  $\phi(\text{val}(\mathcal{P}) - Z) = C$  ( $r$ -SET PACKING); there exists a set  $\mathcal{P} \subseteq \mathcal{S}$  such that  $|\mathcal{P}| \geq \frac{|Z|+|C|}{r}$ , for any  $T, T' \in \mathcal{P}$ ,  $\text{val}(T) \cap \text{val}(T') = \emptyset$ , for each  $T \in \mathcal{P}$  there is a subgraph  $G' \subseteq G[\text{val}(T)]$  that is isomorphic to  $T$ ,  $\text{val}(\mathcal{P}) \cap \text{val}(\mathcal{M}) = Z$ , and  $\phi(\text{val}(\mathcal{P}) - Z) = C$  (GRAPH PACKING); and there exists a set  $\mathcal{P} \subseteq \mathcal{S}$  such that  $|\mathcal{P}| \geq \frac{|Z|+|C|}{r}$ , for any  $T, T' \in \mathcal{P}$ ,  $\text{val}(T) \cap \text{val}(T') = \emptyset$ , for each  $T \in \mathcal{P}$  there is a subgraph  $G'$  that is isomorphic to  $H$  such that each edge of  $G'$  is in  $T$ ,  $\text{val}(\mathcal{P}) \cap \text{val}(\mathcal{M}) = Z$ , and  $\phi(\text{val}(\mathcal{P}) - Z) = C$  (GRAPH EDGE-PACKING). In addition, we alter the conditions for  $P_{\phi}$  to be 1 in a similar manner, though no alteration is needed for  $r$ -SET PACKING; for GRAPH PACKING we add the condition that there is a subgraph  $G' \subseteq G[\text{val}(T)]$  that is isomorphic to  $H$  and for GRAPH EDGE-PACKING we add the condition that there is a subgraph  $G'$  that is isomorphic to  $H$  such that each edge of  $G'$  is in  $T$ .

The analysis of the algorithms depends on Lemma 6 and, in the graph problems the need to find all copies of  $H$  in  $G$ , resulting in the theorem below.

**Theorem 2.** *The problems  $r$ -SET PACKING, GRAPH PACKING, and GRAPH EDGE-PACKING can solved in time  $O(n + 2^{O(rk)})$ ,  $O(n^{|V(H)|} + 2^{O(rk)})$ , and  $O(n^{|E(H)|} + 2^{O(rk)})$ , respectively.*

## 6 Choosing Hash Functions

To construct the family of hash functions  $\mathcal{F} = \{f : U \rightarrow X\}$  with the property that for any subset  $S$  of  $U$  with  $|S| = |X|$ , there is an  $f \in \mathcal{F}$  that is 1-1 on  $S$ , Alon et al. used results from the theory of perfect hash functions together with derandomization of small sample spaces that support almost  $\ell$ -wise independent random variables. They were able to construct a family  $\mathcal{F}$  with  $|\mathcal{F}| = 2^{O(|X|)} \log |U|$ , and this is what we used in Section 4. However, they made no attempt to optimize the constant hidden in the  $O$ -notation, since they were assuming  $|X|$  fixed and  $|U|$  equal to the size of the input.

In our case (and in any practical implementation of the algorithms of Alon et al.), it would be preferable to lower the constant in the exponent as much as possible. We have one advantage: kernelization means that  $|U|$  in our case is  $O(k^r)$ , not  $O(n)$ , and so we are less concerned about dependence on  $|U|$  (note that  $|S| = (r-1)k$  in our case). Two other optimizations are applicable both to our situation and that of Alon et al. First, we can allow more colours, i.e.  $|X| = \alpha(r-1)k$  for some constant  $\alpha$ . This will increase the number of dynamic programming problems, since the number of choices for  $C$  (in determining the number of  $B_{\phi,W}(Z, C)$ ) grows from  $2^{(r-1)k}$  to  $\binom{\alpha(r-1)k}{(r-1)k} \leq (e\alpha)^{(r-1)k}$ . But this may be offset by the reduction in the size of the family of hash functions, since allowing more colours makes it easier to find a perfect hash function. Second,

for some of the work on the theory of perfect hash functions used by Alon et al., it is important that the hash functions be computable in  $O(1)$  time, whereas in our application, we can allow time polynomial in  $k$ .

As an example of applying such optimization, we can make use of the work of Slot and van Emde Boas [SvEB85]. They give a scheme based on the pioneering work of Fredman, Komlós, and Szemerédi [FKS82] that in our case results in a family  $\mathcal{F}$  of size  $2^{4|S|+5\log|S|+3}|U|$ , where  $|X| = 6|S|$ , and it takes  $O(k)$  time to compute the value of a hash function. We will not explore this line of improvement further beyond underlining that there is a tradeoff between the power of the family of hash functions we use and the size, and we have some latitude in choosing families with weaker properties.

Another possibility, also discussed by Alon et al., is to replace the deterministic search for a hash function (in a family where the explicit construction is complicated) by a random choice of colouring. A random  $2|S|$ -colouring of  $U$  is likely to be 1-1 on a subset  $S$  with failure probability that is exponentially small in  $|S|$ . However, this means that a “no” answer has a small probability of error, since it could be due to the failure to find a perfect hash function.

## 7 Conclusions

Our results can be extended to handle problems such as packing or edge-packing graphs from a set of supplied graphs, and more generally to disjoint structures. To express all problems in a general framework, we view tuples as directed hypergraph edges, sets of tuples as hypergraphs, and inputs as triples of a sequence of hypergraphs, a matching-size parameter  $\ell$ , and a supplied hypergraph  $G$ .

To define the necessary terms, we consider the projection of tuples and an associated notion of independence. Given a tuple  $T = (v_1, \dots, v_{r'})$  where  $r' \leq r$  and a subset of indices  $I = (i_1, \dots, i_\ell) \subseteq \{1, \dots, r\}$ , we define the *projection of  $T$  on  $I$*  as  $T[I] = (v_{i_1}, \dots, v_{i_m})$  where  $m = \max\{1, \dots, r'\} \cap \{i_1, \dots, i_\ell\}$ . To form a set of related tuples, given an hypergraph  $\mathcal{S}$  and a subset  $I \subseteq \{1, \dots, r\}$  such that  $|I| = \ell$ , we denote as  $w(\mathcal{S}, I) = \{T' \in V^\ell \mid \text{there exists a tuple } T \in \mathcal{S} \text{ such that } T[I] = T'\}$ . We then say that a subsequence  $\mathcal{C}' \subseteq \mathcal{C}$  is an  $\ell$ -*matching* if for any pair of hypergraphs  $\mathcal{S}, \mathcal{S}' \in \mathcal{C}'$ , for all subsets of indices  $I \subseteq \{1, \dots, r\}$  of size  $\ell$ ,  $w(\mathcal{S}, I) \neq w(\mathcal{S}', I)$ .

Our problem is then one of finding a subsequence  $\mathcal{C}'$  of the hypergraphs forming an  $\ell$ -matching of size at least  $k$  such that the elements of each hypergraph in  $\mathcal{C}'$  can be restricted to a smaller structure isomorphic to  $G$ . This problem subsumes all problems in this paper, and can be solved in time  $O(n + 2^{O(\ell k)})$ .

## References

- [ADP80] G. Ausiello, A. D'Atri, and M. Protasi. Structure preserving reductions among context optimization problems. *Journal of Computer and System Sciences*, 21:136–153, 1980.
- [AYZ95] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the Association for Computing Machinery*, 42(4):844–856, 1995.

- [CFJ04] B. Chor, M. R. Fellows, and D. Juedes. Linear kernels in linear time, or how to save  $k$  colors in  $O(n^2)$  steps,. In *Proceedings of the 30th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2004)*, 2004.
- [CFJK01] J. Chen, D. K. Friesen, W. Jia, and I. Kanj. Using nondeterminism to design deterministic algorithms. In *Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science (FST TCS)*, volume 2245 of *Lecture Notes in Computer Science*, pages 120–131. Springer, 2001.
- [Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual Symposium on the Theory of Computing*, pages 151–158, 1971.
- [Fel03] M. R. Fellows. Blow-ups, win/win’s, and crown rules: Some new directions in FPT. In *Proceedings of the 29th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 03)*, volume 2880 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2003.
- [FHR<sup>+</sup>04] M. R. Fellows, P. Heggenes, F. A. Rosamond, C. Sloper, and J. A. Telle. Exact algorithms for finding  $k$  disjoint triangles in an arbitrary graph. In *Proceedings of the 30th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2004)*, 2004.
- [FKS82] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with  $o(1)$  worst-case access time. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, pages 165–169, 1982.
- [Hol81] I. Holyer. The NP-completeness of some edge-partition problems. *SIAM Journal on Computing*, 10(4):713–717, 1981.
- [JZC04] W. Jia, C. Zhang, and J. Chen. An efficient parameterized algorithm for  $m$ -set packing. *Journal of Algorithms*, 50(1):106–117, 2004.
- [Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [Mar04] D. Marx. Parameterized complexity of constraint satisfaction problems. In *Proceedings of the 19th Annual IEEE Conference on Computational Complexity*, 2004.
- [MPS04] L. Mathieson, E. Prieto, and P. Shaw. Packing edge disjoint triangles: a parameterized view. In *Proceedings of the International Workshop on Parameterized and Exact Computation*, 2004.
- [PS04] E. Prieto and C. Sloper. Looking at the stars. In *Proceedings of the International Workshop on Parameterized and Exact Computation*, 2004.
- [SvEB85] C. F. Slot and P. van Emde Boas. On tape versus core; an application of space efficient perfect hash functions to the invariance of space. *Elektronische Informationsverarbeitung und Kybernetik*, 21(4/5):246–253, 1985.
- [Woe03] G. J. Woeginger. Exact algorithms for NP-hard problems, a survey. In *Combinatorial Optimization – Eureka, You Shrink!*, volume 2570 of *Lecture Notes on Computer Science*, pages 185–207. Springer, 2003.

# On the Evolution of Selfish Routing\*

Simon Fischer and Berthold Vöcking

FB Informatik, LS2, University of Dortmund, 44221 Dortmund, Germany  
{fischer,voecking}@ls2.cs.uni-dortmund.de

**Abstract.** We introduce a model to study the temporal behaviour of selfish agents in networks. So far, most of the analysis of selfish routing is concerned with static properties of equilibria which is one of the most fundamental paradigms in classical Game Theory. By adopting a generalised approach of Evolutionary Game Theory we extend the model of selfish routing to study the dynamical behaviour of agents.

For symmetric games corresponding to singlecommodity flow, we show that the game converges to a Nash equilibrium in a restricted strategy space. In particular we prove that the time for the agents to reach an  $\epsilon$ -approximate equilibrium is polynomial in  $\epsilon$  and only logarithmic in the ratio between maximal and optimal latency. In addition, we present an almost matching lower bound in the same parameters.

Furthermore, we extend the model to asymmetric games corresponding to multicommodity flow. Here we also prove convergence to restricted Nash equilibria, and we derive upper bounds for the convergence time that are linear instead of logarithmic.

## 1 Introduction

Presently, the application of Game Theory to networks and congestion games is gaining a growing amount of interest in Theoretical Computer Science. One of the most fundamental paradigms of Game Theory is the notion of Nash equilibria. So far, many results on equilibria have been derived. Most of these are concerned with the ratio between average cost at an equilibrium and the social optimum, mostly referred to as the *coordination ratio* or the *price of anarchy* [2,7,8,13], ways to improve this ratio, e.g. by taxes [1], and algorithmic complexity and efficiency of computing such equilibria [3,4,5].

Classical Game Theory is based on fully rational behaviour of players and global knowledge of all details of the game under study. For routing games in large networks like the Internet, these assumptions are clearly far away from being realistic. Evolutionary Game Theory makes a different attempt to explain why large populations of agents may or may not “converge” towards equilibrium states. This theory is mainly based on the so-called replicator dynamics, a model of an evolutionary process in which agents revise their strategies from time to time based on local observations.

---

\* Supported in part by the EU within the 6th Framework Programme under contract 001907 (DELIS) and by DFG grant Vo889/1-2.

In this paper, we apply Evolutionary Game Theory to selfish routing. This enables us to study the dynamics of selfish routing rather than only the static structure of equilibria. We prove that the only existing fixed points of the replicator dynamics are Nash equilibria over a restricted strategy space. We prove that these fixed points are “evolutionary stable” which implies that the replicator dynamics converges to one of them. One standard approach in Evolutionary Game Theory to prove stability is based on symmetry properties of payoff matrices. However, we cannot simply cast these results as our model of selfish routing allows for arbitrary latency functions whereas the existing literature on Evolutionary Game Theory assumes affine payoff functions corresponding to linear latency functions with zero offset. In fact our proof of evolutionary stability is based on monotonicity instead of symmetry.

Another aspect that – to our knowledge – has neither been considered in Evolutionary nor in classical Game Theory is the time it takes to reach or come close to equilibria – the speed of convergence. We believe that this is an issue of particular importance as equilibria are only meaningful if they are reached in reasonable time. In fact we can prove that symmetric congestion games – corresponding to singlecommodity flow – converge very quickly to an approximate equilibrium. For asymmetric congestion games our bounds are slightly weaker.

The well established models of selfish routing and Evolutionary Game Theory are described in Section 2. In Section 3 we show how a generalisation of the latter can be applied to the first. In Section 4 we present our results on the speed of convergence. All proofs are collected in Section 5. We finish with some conclusions and open problems.

## 2 Known Models

### 2.1 Selfish Routing

Consider a network  $G = (V, E)$  and for all  $e \in E$ , latency functions  $l_e : [0, 1] \mapsto \mathbb{R}$  assigned to the edges mapping load to latency, and a set of commodities  $\mathcal{I}$ . For commodity  $i \in \mathcal{I}$  there is a fixed flow demand of  $d_i$  that is to be routed from source  $s_i$  to sink  $t_i$ . The total flow demand of all commodities is 1. Denote the set of  $s_i$ - $t_i$ -paths by  $P_i \subseteq \mathcal{P}(E)$ , where  $\mathcal{P}(E)$  is the power set of  $E$ . For simplicity of notation we assume that the  $P_i$  are disjoint, which is certainly true if the pairs  $(s_i, t_i)$  are pairwise distinct. Then there is a unique commodity  $i(p)$  associated with each path  $p \in P$ . Let furthermore  $P = \bigcup_{i \in \mathcal{I}} P_i$ .

For  $p \in P$  denote the amount of flow routed over path  $p$  by  $x_p$ . We combine the individual values  $x_p$  into a vector  $\mathbf{x}$ . The set of legal flows is the simplex<sup>1</sup>  $\Delta := \{\mathbf{x} \mid \forall i \in \mathcal{I} : \sum_{p \in P_i} x_p = d_i\}$ . Furthermore, for  $e \in E$ ,  $x_e := \sum_{p \ni e} x_p$  is the total load of edge  $e$ . The latency of edge  $e \in E$  is  $l_e(x_e)$  and the latency of path  $p$  is

$$l_p(\mathbf{x}) = \sum_{e \in p} l_e(x_e).$$

<sup>1</sup> Strictly speaking,  $\Delta$  is not a simplex, but a product of  $|\mathcal{I}|$  simplices scaled by  $d_i$ .

The average latency with respect to commodity  $i \in \mathcal{I}$  is

$$\bar{l}_i(\mathbf{x}) = d_i^{-1} \sum_{p \in P_i} x_p l_p(\mathbf{x}).$$

In a more general setting one could abstract from graphs and consider arbitrary sets of resources  $E$  and arbitrary non-empty subsets of  $E$  as legal strategies. We do not do this for simplicity, though it should be clear that all our considerations below also hold for the general case.

In order to study the behaviour of users in networks, one assumes that there are an infinite number of agents carrying an infinitesimal amount of flow each. In this context the flow vector  $\mathbf{x}$  can also be seen as a population of agents where  $x_p$  is the non-integral “number” of agents selecting strategy  $p$ .

The individual agents strive to choose a path that minimises their personal latency regardless of social benefit. This does not impute maliciousness to the agents, but simply arises from the lack of a possibility to coordinate strategies. One can ask: What population will arise if we assume complete rationality and if all agents have full knowledge about the network and the latency functions?

This is where the notion of an equilibrium comes into play. A flow or population  $\mathbf{x}$ , is said to be at Nash equilibrium [11], when no agent has an incentive to change their strategy. The classical existence result by Nash holds if mixed strategies are allowed, i. e., agents may choose their strategy by a random distribution. However, mixed strategies and an infinite population of agents using pure strategies are basically the same. A very useful characterisation of a Nash equilibrium is the Wardrop equilibrium [6].

**Definition 1 (Wardrop equilibrium).** *A flow  $\mathbf{x}$  is at Wardrop equilibrium if and only if for all  $i \in \mathcal{I}$  and all  $p, p' \in P_i$  with  $x_p > 0$  it holds that  $l_p(\mathbf{x}) \leq l_{p'}(\mathbf{x})$ .*

Several interesting static aspects of equilibria have been studied, among them the famous *price of anarchy*, which is the ratio between average latency at a Nash equilibrium and optimal average latency.

## 2.2 Evolutionary Game Theory

Classical Game Theory assumes that all agents – equipped with complete knowledge about the game and full rationality – will come to a Wardrop equilibrium. However, these assumptions seem far from realistic when it comes to networks. Evolutionary Game Theory gets rid of these assumptions by modelling the agents’ behaviour in a very natural way that requires the agents simply to observe their own and other agents’ payoff and strategy and change their own strategies based on these observations. Starting with an initial population vector  $\mathbf{x}(0)$  – as a function of time – one is interested in its derivative with respect to time  $\dot{\mathbf{x}}$ .

Originally, Evolutionary Game Theory derives dynamics for large populations of individuals from symmetric two-player games. Any two-player game can be described by a matrix  $\mathbf{A} = (a_{ij})$  where  $a_{ij}$  is the payoff of strategy  $i$  when played against strategy  $j$ . Suppose that two individuals are drawn at random from a

large population to play a game described by the payoff matrix  $\mathbf{A}$ . Let  $x_i$  denote the fraction of players playing strategy  $i$  at some given point of time. Then the expected payoff of an agent playing strategy  $i$  against the opponent randomly chosen from population  $\mathbf{x}$  is the  $i^{th}$  component of the matrix product  $(\mathbf{Ax})_i$ . The average payoff of the entire population is  $\mathbf{x} \cdot \mathbf{Ax}$ , where  $\cdot$  is the inner, or scalar, product.

Consider an initial population in which each agent is assigned a pure strategy. At each point of time, each agent plays against an opponent chosen uniformly at random. The agent observes its own and its opponents payoff and decides to imitate its opponent by adopting its strategy with probability proportional to the payoff difference. One could argue that often it is not possible to observe the opponent's payoff. In that case, consider a random aspiration level for each agent. Whenever an agent falls short of this level, it adopts a randomly observed strategy. Interestingly, both scenarios lead to a dynamics which can be described by the differential equation

$$\dot{x}_i = \lambda(\mathbf{x}) \cdot x_i \cdot ((\mathbf{Ax})_i - \mathbf{xAx}) \quad (1)$$

for some positive function  $\lambda$ . This equation has interesting and desirable properties. First, the growth rate of strategy  $i$  should clearly be proportional to the number of agents already playing this strategy, if we assume homogeneous agents. Then we have  $\dot{x}_i = x_i \cdot g_i(\mathbf{x})$ . Secondly, the growth rate  $g_i(\mathbf{x})$  should increase with payoff, i. e.,  $(\mathbf{Ax})_i > (\mathbf{Ax})_j$  should imply  $g_i(\mathbf{x}) > g_j(\mathbf{x})$  and vice versa. Dynamics having this property are called *monotone*. In order to extend this, we say that  $\mathbf{g}$  is *aggregate monotone* if for inhomogeneous (sub)populations the total growth rate of the subpopulations increases with the average payoff of the subpopulation, i. e., for vectors  $\mathbf{y}, \mathbf{z} \in \Delta$  it holds that  $\mathbf{y} \cdot \mathbf{Ax} < \mathbf{z} \cdot \mathbf{Ax}$  if and only if  $\mathbf{y} \cdot \mathbf{g}(\mathbf{x}) < \mathbf{z} \cdot \mathbf{g}(\mathbf{x})$ . It is known that all aggregate monotone dynamics can be written in the form of equation (1). In Evolutionary Game Theory the most common choice for  $\lambda$  seems to be the constant function 1. For  $\lambda(\mathbf{x}) = 1$ , this dynamics is known as the *replicator dynamics*.

Since the differential equation (1) contains cubic terms there is no general method for solving it. It is found that under some reasonable conditions fixed points of this dynamics coincide with Nash equilibria. For a more comprehensive introduction into Evolutionary Game Theory see for example [14].

### 3 Evolutionary Selfish Flow

We extend the dynamics known from Evolutionary Game Theory to our model of selfish routing. We will see that there is a natural generalisation of equation (1) for our scenario.

#### 3.1 Dynamics for Selfish Routing

First consider symmetric games corresponding to singlecommodity flow. In congestion games latency relates to payoff, but with opposite sign. Unless we have



linear latency functions without offset, we cannot express the payoff by means of a matrix  $\mathbf{A}$ . Therefore we replace the term  $(\mathbf{Ax})_i$  by the latency  $l_i(\mathbf{x})$ . The average payoff  $\mathbf{x} \cdot \mathbf{Ax}$  is replaced by the average latency  $\bar{l}(\mathbf{x})$ . Altogether we have

$$\dot{x}_p = \lambda(\mathbf{x}) \cdot x_p \cdot (\bar{l}(\mathbf{x}) - l_p(\mathbf{x})). \quad (2)$$

Evolutionary Game Theory also allows for asymmetric games, i. e., games where each agent belongs to one class determining the set of legal strategies. In principle, asymmetric games correspond to multicommodity flow. The suggested generalisations for asymmetric games, however, are not particularly useful in our context since they assume that agents of the same class do not play against each other. We suggest a simple and natural generalisation towards multicommodity flow. Agents behave exactly as in the singlecommodity case but they compare their own latency to the average latency over the agents in the same class. Although in Evolutionary Game Theory  $\lambda = 1$  seems to be the most common choice even for asymmetric games, we suggest to choose the factor  $\lambda$  dependent on the commodity. We will later see why this might be useful. This gives the following variant of the replicator dynamics:

$$\dot{x}_p = \lambda_{i(p)}(\mathbf{x}) \cdot x_p \cdot (\bar{l}_{i(p)}(\mathbf{x}) - l_p(\mathbf{x})). \quad (3)$$

We believe that this, in fact, is a realistic model of communication in networks since agents only need to “communicate” with agents having the same source and destination nodes.

In order for equation (3) to constitute a legal dynamics, we must ensure that for all  $t$  the population shares sum up to the total flow demand.

**Proposition 1.** *Let  $\mathbf{x}(t)$  be a solution of equation (3). Then  $\mathbf{x}(t) \in \Delta$  for all  $t$ .*

This is proved in Section 5. Note that strategies that are not present in the initial population are not generated by the dynamics. Conversely, positive strategies never get completely extinct.

### 3.2 Stability and Convergence

Maynard Smith and Price [9] introduced the concept of evolutionary stability which is stricter than the concept of a Nash equilibrium. Intuitively, a strategy is evolutionary stable if it is at a Nash equilibrium and earns more against a mutant strategy than the mutant strategy earns against itself. Adopted to selfish routing we can define it in the following way.

**Definition 2 (evolutionary stable).** *A strategy  $\mathbf{x}$  is evolutionary stable if it is at a Nash equilibrium and  $\mathbf{x} \cdot \mathbf{l}(\mathbf{y}) < \mathbf{y} \cdot \mathbf{l}(\mathbf{y})$  for all best replies  $\mathbf{y}$  to  $\mathbf{x}$ .*

A strategy  $\mathbf{y}$  is a *best reply* to strategy  $\mathbf{x}$  if no other strategy yields a better payoff when played against  $\mathbf{x}$ .

**Proposition 2.** *Suppose that latency functions are strictly increasing. If  $\mathbf{x}$  is at a Wardrop equilibrium, then  $\mathbf{x} \cdot \mathbf{l}(\mathbf{y}) < \mathbf{y} \cdot \mathbf{l}(\mathbf{y})$  for all  $\mathbf{y}$  (especially, for all best replies) and hence  $\mathbf{x}$  is evolutionary stable.*



Consider a Wardrop equilibrium  $\mathbf{x}$  and a population  $\mathbf{y}$  such that for some strategy  $p \in P$ ,  $y_p = 0$  and  $x_p > 0$ . The replicator dynamics starting at  $\mathbf{y}$  does not converge to  $\mathbf{x}$  since it ignores strategy  $p$ . Therefore we consider a *restricted strategy space*  $P'$  containing only the paths with positive value and *restricted Wardrop equilibria* over this restricted strategy space.

**Proposition 3.** *Suppose that for all  $i, j \in \mathcal{I}$ ,  $\lambda_i = \lambda_j$  and  $\lambda_i(\mathbf{x}) \geq \epsilon$  for some  $\epsilon > 0$  and any  $\mathbf{x} \in \Delta$ . Let  $\mathbf{y}(t)$  be a solution to the replicator dynamics (3) and let  $\mathbf{x}$  be a restricted Wardrop equilibrium with respect to  $\mathbf{y}(0)$ . Then  $\mathbf{y}$  converges towards  $\mathbf{x}$ , i. e.,  $\lim_{t \rightarrow \infty} \|\mathbf{y}(t) - \mathbf{x}\| = 0$ .*

Both propositions are proved in Section 5.

## 4 Speed of Convergence

In order to study the speed of convergence we suggest two modifications to the standard approach of Evolutionary Game Theory. First, one must ensure that the growth rate of the population shares does not depend on the scale by which we measure latency, as is the case if we choose  $\lambda(\mathbf{x}) = 1$  or any other constant. Therefore we suggest to choose  $\lambda_i(\mathbf{x}) = \bar{l}_i(\mathbf{x})^{-1}$ . This choice arises quite naturally if we assume that the probability of agents changing their strategy depends on their relative latency with respect to the current average latency. We call the resulting dynamics the *relative replicator dynamics*.

Secondly, the Euclidian distance  $\|\mathbf{y} - \mathbf{x}\|$  is not a suitable measure for approximation. Since “sleeping minorities on cheap paths” may grow arbitrarily slowly, it may take arbitrarily long for the current population to come close to the final Nash equilibrium in Euclidian distance. The idea behind our definition of approximate equilibria is not to wait for these sleeping minorities.

**Definition 3 ( $\epsilon$ -approximate equilibrium).** *Let  $P_\epsilon$  be the set of paths that have latency at least  $(1 + \epsilon) \cdot \bar{l}$ , i. e.,  $P_\epsilon = \{p \in P \mid l_p(\mathbf{x}) \geq (1 + \epsilon) \cdot \bar{l}\}$  and let  $x_\epsilon := \sum_{p \in P_\epsilon} x_p$  be the number of agents using these paths. A population  $\mathbf{x}$  is said to be at an  $\epsilon$ -approximate equilibrium if and only if  $x_\epsilon \leq \epsilon$ .*

Note that, by our considerations above,  $\epsilon$ -approximate equilibria can be left again, when minorities start to grow.

We will give our bounds in terms of maximal and optimal latency. Denote the maximum latency by  $l_{\max} := \max_{p \in P} l_p(\mathbf{e}_p)$  where  $\mathbf{e}_p$  is the unit vector for path  $p$ . Let  $l^* := \min_{\mathbf{x} \in \Delta} \bar{l}(\mathbf{x})$  be the average latency at a social optimum.

**Theorem 1.** *The replicator dynamics for general singlecommodity flow networks and non-decreasing latency functions converges to an  $\epsilon$ -approximate equilibrium within time  $\mathcal{O}(\epsilon^{-3} \cdot \ln(l_{\max}/l^*))$ .*

This theorem is robust. The proof does not require that agents behave exactly as described by the replicator dynamics. It is only necessary that a constant fraction of the agents that are by a factor of  $(1 + \epsilon)$  above the average move to a better strategy. We can also show that our bound is in the right ballpark.

**Theorem 2.** *For any  $r := l_{\max}/l^*$  there exists a network and boundary conditions such that the time to reach an  $\epsilon$ -approximate equilibrium is bounded from below by  $\Omega(\epsilon^{-1} \ln r)$ .*

For the multicommodity case, we can only derive a linear upper bound. If we define  $P_\epsilon := \{p | l_p \geq (1 + \epsilon) \cdot \bar{l}_{i(p)}\}$  then the definition of  $x_\epsilon$  and Definition 3 translate naturally to the multicommodity case. Let  $l_i^* = \min_{\mathbf{x} \in \Delta} \bar{l}_i(\mathbf{x})$  be the minimal average latency for commodity  $i \in \mathcal{I}$  and let  $l^* = \min_{i \in \mathcal{I}} l_i^*$ .

**Theorem 3.** *The multicommodity replicator dynamics converges to an  $\epsilon$ -approximate equilibrium within time  $\mathcal{O}(\epsilon^{-3} \cdot l_{\max}/l^*)$ .*

Our analysis of convergence in terms of  $\epsilon$ -approximate equilibria uses Rosenthal's potential function [12]. However, this function is not suitable for the proof of convergence in terms of the Euclidian distance since it does not give a general upper bound. Here we use the entropy as a potential function, which in turn is not suitable for  $\epsilon$ -approximations. Because of our choice of the  $\lambda_i$  in this section, Proposition 3 cannot be translated directly to the multicommodity case.

## 5 Proofs

First we prove that the relative replicator dynamics is a legal dynamics in the context of selfish routing, i.e., it does not leave the simplex  $\Delta$ .

*Proof (of Proposition 1).* We show that for all commodities  $i \in \mathcal{I}$  the derivatives  $\dot{x}_p$ ,  $p \in P_i$  sum up to 0.

$$\begin{aligned} \sum_{p \in P_i} \dot{x}_p &= \lambda_i(\mathbf{x}) \left( \sum_{p \in P_i} x_p \bar{l}_i(\mathbf{x}) - \sum_{p \in P_i} x_p l_p(\mathbf{x}) \right) \\ &= \lambda_i(\mathbf{x}) \left( \bar{l}_i(\mathbf{x}) \sum_{p \in P_i} x_p - d_i \cdot d_i^{-1} \sum_{p \in P_i} x_p l_p(\mathbf{x}) \right) \\ &= \lambda_i(\mathbf{x}) (\bar{l}_i(\mathbf{x}) \cdot d_i - d_i \cdot \bar{l}_i(\mathbf{x})) = 0. \end{aligned}$$

Therefore,  $\sum_{p \in P_i} x_p = d_i$  is constant. □

Now we prove evolutionary stability.

*Proof (of Proposition 2).* Let  $\mathbf{x}$  be a Nash equilibrium. We want to show that  $\mathbf{x}$  is evolutionary stable. In a Wardrop equilibrium all latencies of used paths belonging to the same commodity are equal. The latency of unused paths is equal or even greater than the latency of used paths. Therefore  $\mathbf{x} \cdot \mathbf{l}(\mathbf{x}) \leq \mathbf{y} \cdot \mathbf{l}(\mathbf{x})$  for all populations  $\mathbf{y}$ . As a consequence,

$$\begin{aligned} \mathbf{y} \cdot \mathbf{l}(\mathbf{y}) &\geq \mathbf{x} \cdot \mathbf{l}(\mathbf{x}) + \mathbf{y} \cdot \mathbf{l}(\mathbf{y}) - \mathbf{y} \cdot \mathbf{l}(\mathbf{x}) \\ &= \mathbf{x} \cdot \mathbf{l}(\mathbf{x}) + \sum_{p \in P} y_p (l_p(\mathbf{y}) - l_p(\mathbf{x})) \\ &= \mathbf{x} \cdot \mathbf{l}(\mathbf{x}) + \sum_{e \in E} y_e (l_e(\mathbf{y}) - l_e(\mathbf{x})). \end{aligned}$$

Consider an edge  $e \in E$ . There are three cases.

1.  $y_e > x_e$ . Because of strict monotonicity of  $l_e$ , it holds that  $l_e(\mathbf{y}) > l_e(\mathbf{x})$ . Therefore also  $y_e(l_e(\mathbf{y}) - l_e(\mathbf{x})) > x_e(l_e(\mathbf{y}) - l_e(\mathbf{x}))$ .
2.  $y_e < x_e$ . Because of strict monotonicity of  $l_e$ , it holds that  $l_e(\mathbf{y}) < l_e(\mathbf{x})$ . Again,  $y_e(l_e(\mathbf{y}) - l_e(\mathbf{x})) > x_e(l_e(\mathbf{y}) - l_e(\mathbf{x}))$ .
3.  $y_e = x_e$ . In that case  $y_e(l_e(\mathbf{y}) - l_e(\mathbf{x})) = x_e(l_e(\mathbf{y}) - l_e(\mathbf{x}))$ .

There is at least one edge  $e \in E$  with  $x_e \neq y_e$  and, therefore,  $y_e(l_e(\mathbf{y}) - l_e(\mathbf{x})) > x_e(l_e(\mathbf{y}) - l_e(\mathbf{x}))$ . Altogether we have

$$\mathbf{y} \cdot \mathbf{l}(\mathbf{y}) > \mathbf{x} \cdot \mathbf{l}(\mathbf{x}) + \sum_{e \in E} x_e(l_e(\mathbf{y}) - l_e(\mathbf{x})) = \mathbf{x} \cdot \mathbf{l}(\mathbf{y})$$

which is our claim. Note that this proof immediately covers the multicommodity case.  $\square$

*Proof (of Proposition 3).* Denote the Nash equilibrium by  $\mathbf{x}$  and the current population by  $\mathbf{y}$ . We define a potential function  $H_{\mathbf{x}}$  by the entropy

$$H_{\mathbf{x}}(\mathbf{y}) := \sum_{p \in P} x_p \ln \frac{x_p}{y_p}.$$

From information theory it is known that this function always exceeds the square of the Euclidean distance  $\|\mathbf{x} - \mathbf{y}\|^2$ . We can also write this as  $H_{\mathbf{x}}(\mathbf{y}) = \sum_{p \in P} (x_p \ln(x_p) - x_p \ln(y_p))$ . Using the chain rule we calculate the derivative with respect to time:

$$\dot{H}_{\mathbf{x}}(\mathbf{y}) = - \sum_{p \in P} x_p \dot{y}_p \frac{1}{y_p}.$$

Now we substitute the replicator dynamics for  $\dot{y}_p$ , cancelling out the  $y_p$ .

$$\begin{aligned} \dot{H}_{\mathbf{x}}(\mathbf{y}) &= \lambda(\mathbf{y}) \sum_{i \in \mathcal{I}} \sum_{p \in P_i} x_p (l_p(\mathbf{y}) - \bar{l}_i(\mathbf{y})) \\ &= \lambda(\mathbf{y}) \sum_{i \in \mathcal{I}} \left( \sum_{p \in P_i} x_p l_p(\mathbf{y}) - \bar{l}_i(\mathbf{y}) \cdot d_i \right) \\ &= \lambda(\mathbf{y}) \sum_{i \in \mathcal{I}} \left( \sum_{p \in P_i} x_p l_p(\mathbf{y}) - \left( d_i^{-1} \sum_{p \in P_i} y_p l_p(\mathbf{y}) \right) \cdot d_i \right) \\ &= \lambda(\mathbf{y}) \sum_{i \in \mathcal{I}} \sum_{p \in P_i} (x_p - y_p) \cdot l_p(\mathbf{y}) \\ &= \lambda(\mathbf{y}) \cdot (\mathbf{x} - \mathbf{y}) \cdot \mathbf{l}(\mathbf{y}). \end{aligned}$$

Since  $\mathbf{x}$  is a Wardrop equilibrium, Proposition 2 implies that  $(\mathbf{x} - \mathbf{y}) \cdot \mathbf{l}(\mathbf{y}) < 0$ . Furthermore, by our assumption  $\lambda(\mathbf{x}) \geq \epsilon > 0$ . Altogether this implies that  $H_{\mathbf{x}}(\mathbf{y})$ , and therefore also  $\|\mathbf{x} - \mathbf{y}\|^2$  decreases towards the lower bound of  $H_{\mathbf{x}}$ , which is 0.  $\square$

We will now proof the bound on the time of convergence in the symmetric case.

*Proof (of Theorem 1).* For our proof we will use a generalisation of Rosenthal's potential function [12]. In the discrete case, this potential function inserts the agents sequentially into the network and sums up the latencies they experience at the point of time they are inserted. In the continuous case this sum can be generalised to an integral. As a technical trick, we furthermore add the average latency at a social optimum  $l^*$ :

$$\Phi(\mathbf{x}) := \left( \sum_{e \in E} \int_0^{x_e} l_e(x) dx \right) + l^*. \quad (4)$$

Now we calculate the derivative with respect to time of this potential  $\Phi$ . Let  $L_e$  be an antiderivative of  $l_e$ .

$$\dot{\Phi} = \sum_{e \in E} \dot{L}_e(x_e) = \sum_{e \in E} \dot{x}_e \cdot l_e(x_e) = \sum_{e \in E} \sum_{p \ni e} \dot{x}_p \cdot l_e(x_e).$$

Now we substitute the replicator dynamics (2) into this equation and obtain

$$\begin{aligned} \dot{\Phi} &= \sum_{e \in E} \sum_{p \ni e} (\lambda(\mathbf{x}) \cdot x_p \cdot (\bar{l}(\mathbf{x}) - l_p(\mathbf{x}))) \cdot l_e(x_e) \\ &= \lambda(\mathbf{x}) \sum_{p \in P} \sum_{e \in p} x_p \cdot (\bar{l}(\mathbf{x}) - l_p(\mathbf{x})) \cdot l_e(x_e) \\ &= \lambda(\mathbf{x}) \sum_{p \in P} x_p \cdot (\bar{l}(\mathbf{x}) - l_p(\mathbf{x})) \cdot l_p(\mathbf{x}) \\ &= \lambda(\mathbf{x}) \left( \bar{l}(\mathbf{x}) \sum_{p \in P} x_p l_p(\mathbf{x}) - \sum_{p \in P} x_p l_p(\mathbf{x})^2 \right) \\ &= \lambda(\mathbf{x}) \left( \bar{l}(\mathbf{x})^2 - \sum_{p \in P} x_p l_p(\mathbf{x})^2 \right). \end{aligned} \quad (5)$$

By Jensen's inequality this difference is negative.

As long as we are not at an  $\epsilon$ -approximate equilibrium, there must be a population share of magnitude at least  $\epsilon$  with latency at least  $(1 + \epsilon) \cdot \bar{l}(\mathbf{x})$ . For fixed  $\bar{l}(\mathbf{x})$  the term  $\sum_{p \in P} x_p l_p(\mathbf{x})^2$  is minimal when the less expensive paths all have equal latency  $l'$ . This follows from Jensen's inequality as well. We have  $\bar{l} = \epsilon \cdot (1 + \epsilon) \cdot \bar{l} + (1 - \epsilon) \cdot l'$  and

$$l' = \bar{l} \cdot \frac{1 - \epsilon - \epsilon^2}{1 - \epsilon}. \quad (6)$$

According to equation (5) we have

$$\dot{\Phi} = \lambda(\mathbf{x}) \cdot (\bar{l}(\mathbf{x})^2 - (\epsilon \cdot ((1 + \epsilon) \cdot \bar{l}(\mathbf{x}))^2 + (1 - \epsilon) \cdot l'^2)). \quad (7)$$

Substituting (6) into (7) and doing some arithmetics we get

$$\begin{aligned}\dot{\Phi} &= -\lambda(\mathbf{x}) \frac{\epsilon^3}{1-\epsilon} \bar{l}(\mathbf{x})^2 \\ &\leq -\lambda(\mathbf{x}) \cdot \epsilon^3 \cdot \bar{l}(\mathbf{x})^2 / 2 = -\epsilon^3 \cdot \bar{l}(\mathbf{x}) / 2.\end{aligned}\tag{8}$$

We can bound  $\bar{l}$  from below by  $\Phi/2$ :

$$\begin{aligned}\bar{l}(\mathbf{x}) &= \sum_{p \in P} x_p l_p(\mathbf{x}) = \sum_{p \in P} \sum_{e \in p} x_p l_e(x_e) \\ &= \sum_{e \in E} \sum_{p \ni e} x_p l_e(x_e) = \sum_{e \in E} x_e l_e(x_e) \\ &\geq \sum_{e \in E} \int_0^{x_e} l_e(x) dx\end{aligned}\tag{9}$$

The inequality holds because of monotonicity of the latency functions. By definition of  $l^*$ , also  $\bar{l} \geq l^*$ . Altogether we have  $\bar{l} + \bar{l} \geq l^* + \sum_{e \in E} \int_0^{x_e} l_e(x) dx$ , or  $\bar{l} \geq \Phi/2$ . Substituting this into inequality (8) we get the differential inequality

$$\dot{\Phi} \leq -\epsilon^3 \Phi / 4$$

which can be solved by standard methods. It is solved by any function

$$\Phi(t) \leq \Phi_{init} e^{-\epsilon^3/4 \cdot t}.$$

where  $\Phi_{init} = \Phi(0)$  is given by the boundary conditions. This does only hold as long as we are not at a  $\epsilon$ -approximate equilibrium. Hence, we must reach a  $\epsilon$ -approximate equilibrium at the latest when  $\Phi$  falls below its minimum  $\Phi^*$ . We find that the smallest  $t$  fulfilling  $\Phi(t) \leq \Phi^*$  is

$$t = 4\epsilon^{-3} \ln \frac{\Phi_{init}}{\Phi^*}.$$

Clearly,  $\Phi^* \geq l^*$  and  $\Phi_{init} \leq 2 \cdot l_{max}$  which establishes our assertion.  $\square$

Where is this proof pessimistic? There are only two estimates. We will give an example where these inequalities almost hold with equality.

1. Inequality (9) holds with equality if we use constant latency functions.
2. The considerations leading to equation (7) are pessimistic in that they assume that there are always very few agents in  $x_e$  and that they are always close to  $\bar{l}$ .

Starting from these observations we can construct a network in which we can prove our lower bound.

*Proof (of Theorem 2, Sketch).* Our construction is by induction. Consider a network with  $m$  parallel links numbered 0 through  $m-1$ . We show that the time of convergence for  $m$  links is at least  $(m-1) \cdot \Omega(\epsilon^{-1})$ .

For  $i \in \{1 \dots, m\}$  define the (constant) latency functions  $l_i(x) = (1+c\epsilon)^{-i+1}$ ,  $c$  a constant large enough. Initially, some agents are on the most expensive link 1, very few agents are on links  $3, \dots, m$ , and the majority is on link 2. More precisely,  $x_1(0) = 2\epsilon$ ,  $x_2(0) = 1 - 2\epsilon - \gamma$ , and  $\sum_{i=3}^m x_i(0) = \gamma$ , where  $\gamma$  is some small constant. In the induction step we will define the values of the  $x_i(0)$  for  $i > 2$ . Initially, assume  $\gamma = 0$ .

First consider the case where  $m = 2$ . Clearly, the average latency  $\bar{l}$  is dominated by link 2, i.e.,  $l_1$  is by a factor of at least  $(1 + \epsilon)$  more expensive than  $\bar{l}$ . This implies that we cannot be at an  $\epsilon$ -approximate equilibrium as long as  $x_1(t) > \epsilon$ . Since link 1 is by a factor of  $\Theta(1 + \epsilon)$  more expensive than  $\bar{l}$  we have  $\dot{x}_1 = -\Theta(\epsilon^2)$  and it takes time  $\Omega(\epsilon^{-1})$  for  $x_1$  to decrease from  $2\epsilon$  to  $\epsilon$ .

Now consider a network with  $m > 2$  edges. By induction hypothesis we know that it takes time  $t_m = (m-2) \cdot \Omega(\epsilon^{-1})$  for the agents to shift their load to link  $m-1$ . We carefully select  $x_m(0)$  such that it fulfils the following conditions:

- For  $t < t_m$  the population share on link  $m$  does not lower  $\bar{l}$  significantly such that our assumptions on the growth rates still hold.
- For  $t = t_m$ ,  $x_m(t)$  exceeds a threshold such that  $\bar{l}$  decreases below  $l_{m-1}(1+\epsilon)$ .

Although we do not calculate it, there surely exists a boundary condition for  $x_m(0)$  having these properties. Because of the second condition, the system exactly fails to enter an  $\epsilon$ -approximate equilibrium at time  $t_m$ . Because of this, we must wait another phase of length  $\Omega(\epsilon^{-1})$  for the agents on link  $m-1$  to switch to link  $m$ . Note that there may be agents on link  $m-2$  and above. These move faster towards cheaper links, but this does not affect our lower bound on the agents on link  $m-1$ .

We have  $l_{\max} = 1$  and  $l^* = (1 + c\epsilon)^{-m+1}$ . For arbitrary ratios  $r = l_{\max}/l^*$  we choose  $m = \ln(r)$  yielding a lower bound of  $\Omega(\epsilon^{-1} \cdot m) = \Omega(\epsilon^{-1} \cdot \ln r)$  on the time of convergence.  $\square$

*Proof (of Theorem 3).* In the multicommodity case, equation (5) takes the form

$$\dot{\Phi} = \sum_{i \in \mathcal{I}} \lambda_i(\mathbf{x}) \left( \bar{l}_i^2 - \sum_{p \in P_i} x_p l_p(\mathbf{x})^2 \right).$$

Let  $i^* = \arg \min_{i \in \mathcal{I}} \bar{l}_i$ . In the worst case, all agents in  $x_\epsilon$  belong to commodity  $i^*$  and equation (8) takes the form  $\dot{\Phi} \leq -\epsilon^{-3} \bar{l}_{i^*}/2$ . Then Theorem 3 follows directly by the trivial estimate  $\bar{l}_{i^*} > l^*$ .  $\square$

## 6 Conclusions and Open Problems

We introduced the replicator dynamics as a model for the dynamic behaviour of selfish agents in networks. For the symmetric case we have given an essentially tight bound for the time of convergence of this dynamics that is polynomial in the degree of approximation and logarithmic in network parameters. For the

multicommodity case, we derived an upper bound which is linear in the network parameters. This model can also be used in the design of distributed load balancing algorithms, since one can reasonably assume that an algorithm based on this model would be accepted by network users. Several interesting problems remain open:

- The replicator dynamics is based on random experiments performed by agents playing against each other. By this “fluid limit” we can ensure that the outcomes of the experiments meet their expectation values. What happens if we go back to the discrete process?
- How can one improve the upper bound in the multicommodity scenario?
- There is a delay between the moment the agents observe load and latency in the network and the moment they actually change their strategy. What effects does the use of old information have? Similar questions are, e.g., studied in [10].
- A question of theoretical interest is: What is the convergence time for the final Nash equilibrium?

## References

1. Richard Cole, Yevgeniy Dodis, and Tim Roughgarden. Pricing network edges for heterogeneous selfish users. In *STOC 2003*, pages 521–530, 2003.
2. Arthur Czumaj and Vöcking Berhold. Tight bounds for worst-case equilibria. In *Proc. 13th SODA*, pages 413–420, San Francisco, 2002.
3. Alex Fabrikant, Christos Papadimitriou, and Kunal Talwar. The complexity of pure Nash equilibria, 2004. to appear.
4. D. Fotakis, S. Kontogiannis, E. Koutsoupias, M. Mavronicolas, and P. Spirakis. The structure and complexity of Nash equilibria for a selfish routing game. In *Proc. of the ICALP*, pages 123–134, Malaga, Spain, 2002.
5. Martin Gairing, Thomas Luecking, Marios Mavronicolas, and Burkhard Monien. Computing Nash equilibria for scheduling on restricted parallel links. In *Proc. of the STOC 2004*, 2004.
6. Alain B. Haurie and Patrice Marcotte. On the relationship between Nash-Cournot and Wardrop equilibria. *Networks*, 15:295–308, 1985.
7. E. Koutsoupias and C. H. Papadimitriou. Worst-case equilibria. In *STACS '99*, 1999.
8. Marios Mavronicolas and Paul G. Spirakis. The price of selfish routing. In *Proc. of STOC 2001*, pages 510–519, 2001.
9. J. Maynard Smith and G. R. Price. The logic of animal conflict. *Nature*, 246:15–18, 1973.
10. Michael Mitzenmacher. How useful is old information? In *Proc. of the PODC 1997*, pages 83–91, 1997.
11. John F. Nash. Equilibrium points in  $n$ -person games. In *Proc. of National Academy of Sciences*, volume 36, pages 48–49, 1950.
12. Robert W. Rosenthal. A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory*, 2:65–67, 1973.
13. Tim Roughgarden and Eva Tardos. How bad is selfish routing? *J. ACM*, 49(2):236–259, 2002.
14. Jörgen W. Weibull. *Evolutionary Game Theory*. MIT press, 1995.

# Competitive Online Approximation of the Optimal Search Ratio\*

Rudolf Fleischer<sup>1\*\*</sup>, Tom Kamphans<sup>2</sup>, Rolf Klein<sup>2</sup>, Elmar Langetepe<sup>2</sup>, and  
Gerhard Trippen<sup>3\*\*</sup>

<sup>1</sup> fleischer@acm.org.

<sup>2</sup> University of Bonn, Institute of Computer Science I, D-53117 Bonn, Germany.  
[kamphans,rolf.klein,langetep]@informatik.uni-bonn.de.

<sup>3</sup> The Hong Kong University of Science and Technology, CS Dept., Hong Kong.  
trippen@cs.ust.hk.

**Abstract.** How efficiently can we search an unknown environment for a goal in unknown position? How much would it help if the environment were known? We answer these questions for simple polygons and for general graphs, by providing online search strategies that are as good as the best offline search algorithms, up to a constant factor. For other settings we prove that no such online algorithms exist.

## 1 Introduction

One of the recurring tasks in life is to search one's environment for an object whose location is—at least temporarily—unknown. This problem comes in different variations. The searcher may have vision, or be limited to sensing by touch. The environment may be a simple polygon, e. g., an apartment, or a graph, like a street network. Finally, the environment may be known to the searcher, or be unknown.

Such search problems have attracted a lot of interest in online motion planning, see for example the survey by Berman [4]. Usually the cost of a search is measured by the length of the search path traversed; this in turn is compared against the length of the shortest path from the start position to the point where the goal is reached. If we are searching in an unknown environment, the maximum quotient, over all goal positions and all environments, is the *competitive ratio* of the search algorithm.

Most prominent is the problem of searching two half-lines emanating from a common start point. The “doubling” strategy visits the half-lines alternatingly,

---

\* The work described in this paper was partially supported by a grant from the Germany/Hong Kong Joint Research Scheme sponsored by the Research Grants Council of Hong Kong and the German Academic Exchange Service (Project No. G-HK024/02).

\*\* The authors were partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. HKUST6010/01E) and by the RGC Direct Allocation Grant DAG03/04.EG05.



each time doubling the depth of exploration. This way, the goal point is reached after traversing a path at most 9 times as long as its distance from the start, and the competitive ratio of 9 is optimal for this problem; see Baeza-Yates et al. [3] and Alpern and Gal [2]. This doubling approach frequently appears as a subroutine in more complex navigation strategies.

In searching  $m > 2$  half-lines, a constant ratio with respect to the distance from the start can no longer be achieved. Indeed: Even if the half lines were replaced by segments of the same finite length, the goal could be placed at the end of the segment visited last, causing the ratio to be at least  $2m - 1$ . Exponentially increasing the exploration depth by  $m/m - 1$  is known to lead to an optimal competitive ratio of  $C(m) = 1 + 2m(\frac{m}{m-1})^{m-1} \leq 1 + 2me$ .

Much less is known about more realistic settings. Suppose a searcher with vision wants to search an unknown simple polygon for a goal in unknown position. He could employ the  $m$ -way technique from above: By exploring the shortest paths from the start to the  $m$  reflex vertices of the polygon—ignoring their tree structure—a competitive ratio of  $C(m)$  can easily be achieved [13]. Schuierer [15] has refined this method and obtained a ratio of  $C(2k)$ , where  $k$  denotes the smallest number of convex and concave chains into which the polygon's boundary can be decomposed.

But these results do not completely settle the problem. For one, it is not clear why the numbers  $m$  or  $k$  should measure the difficulty of searching a polygon. Also, human searchers easily outperform  $m$ -way search, because they make educated guesses about the shape of those parts of the polygon not yet visited.

In this paper we take the following approach: Let  $\pi$  be a *search path* for the fixed polygon  $P$ , i. e., a path from the start point,  $s$ , through  $P$  from which each point  $p$  inside  $P$  will eventually be visible. Let  $\text{rise}_\pi(p)$  be the first point on  $\pi$  where this happens. The cost of getting to  $p$  via  $\pi$  equals the length of  $\pi$  from  $s$  to  $\text{rise}_\pi(p)$ , plus the Euclidean distance from  $\text{rise}_\pi(p)$  to  $p$ . We divide this value by the length of the shortest  $s$ -to- $p$  path in  $P$ . The maximum of these ratios, over all  $p \in P$ , is the *search ratio* of  $\pi$ . The lowest search ratio possible, over all search paths, is the *optimum search ratio* of  $P$ ; it measures the “searchability” of  $P$ .

Koutsoupias et al. [14] studied graphs with unit length edges where the goal can only be located at vertices, and they only studied the offline case, i. e., with full a priori knowledge of the graph. They showed that computing the optimal search ratio offline is an NP-complete problem, and gave a polynomial time 8-approximation algorithm based on the doubling heuristic.

The crucial question we are considering in this paper is the following: Is it possible to design an *online* search strategy whose search ratio stays within a constant factor of the optimum search ratio, for arbitrary instances of the environment? Surprisingly, the answer is positive for simple polygons as well as for general graphs. (However, for polygons with holes, and for graphs with unit edge length, where the goal positions are restricted to the vertices, no such online strategy exists.)

Note that *search ratio* and *competitive ratio* have very similar definitions, but they are actually rather different concepts. For the competitive ratio, an

online search algorithm has no idea how the environment looks like and has to learn it while searching for the goal. In contrast, the search ratio is defined for a given fixed environment. Since the optimal search ratio path minimizes the quotient of search distance over shortest distance to the goal, the optimal search ratio is actually a lower bound for the competitive search ratio of any online search algorithm. Computing online a  $c$ -approximation of the optimal search ratio path means we compute online a search path whose competitive ratio is at most a factor of  $c$  worse than the optimal competitive ratio of any online search algorithm, but that does not tell us anything about the competitive ratio itself which could be arbitrarily bad. In some special cases, we can search with a constant competitive ratio, for example on the line optimally 9-competitive (the Lost-Cow problem [3]) and in street polygons optimally  $\sqrt{2}$ -competitive [12,16].

The *search* strategies we will present use, as building blocks, modified versions of constant-competitive strategies for online *exploration*, namely the exploration strategy by Hoffmann et al. [11] for simple polygons, and the tethered graph exploration strategy by Duncan et al. [9].

At first glance it seems quite natural to employ an exploration strategy in searching—after all, either task involves looking at each point of the environment. But there is a serious difference in performance evaluation! In searching an environment, we compete against shortest start-to-goal paths, so we have to proceed in a BFS manner. In exploration, we are up against the shortest round trip from which each point is visible; this means, once we have entered some remote part of the environment we should finish it, in a DFS manner, before moving on. However, we can fit these exploration strategies to our search problem by restricting them to a bounded part of the environment. This will be shown in Section 3 where we present our general framework, which turns out to be quite elegant despite the complex definitions. The framework can be applied to online and offline search ratio approximations. In Section 2 we review basic definitions and notations. Our framework will then be applied to searching in various environments like trees, (planar) graphs, and (rectilinear) polygonal environments with and without holes in Sections 4 and 5. Finally, in Section 6, we conclude with a summary of our results (see also Table 1).

## 2 Definitions

We want to find a good search path in some given *environment*  $\mathcal{E}$ . This might be a tree, a (planar) graph, or a (rectangular) polygon with or without holes. In a graph environment, the edges may either have unit length or arbitrary length. Edge lengths do not necessarily represent Euclidean distances, not even in embedded planar graphs. In particular, we do not assume that the triangle inequality holds.

The *goal set*,  $\mathcal{G}$ , is the set of locations in the environment where the (stationary!) goal might be hidden. If  $\mathcal{E}$  is a graph  $G = (V, E)$ , then the goal may be located on some edge (*geometric search*), i. e.,  $\mathcal{G} = V \cup E$ , or its position may be restricted to the vertices (*vertex search*), i. e.,  $\mathcal{G} = V$ . To *explore*  $\mathcal{E}$  means to

**Table 1.** Summary of our approximation results, where  $\alpha > 0$  is an arbitrary constant. The entry marked with \* had earlier been proven by Koutsoupias et al. [14]. They had also shown that computing the optimal search path is NP-complete for (planar) graphs. It is also NP-complete for polygons, whereas it is not known to be NP-complete for trees.

Environment	Edge length	Goal	Polytime approximation ratio	
			Online	Offline
Tree	unit, arbitrary	vertex, geometric	4	4
Planar graph	arbitrary	vertex	no search-competitive alg.	8
Planar graph	unit	vertex	$104 + 40\alpha + \frac{64}{\alpha}$	4
General graph	unit	vertex	no search-competitive alg.	8*
General graph	arbitrary	geometric	$48 + 16\alpha + \frac{32}{\alpha}$	4
Simple polygon			212	8
Rect. simple polygon			$8\sqrt{2}$	8
Polygon with rect. holes			no search-competitive alg.	?

move around in  $\mathcal{E}$  until all potential goal positions  $\mathcal{G}$  have been seen. To *search*  $\mathcal{E}$  means to follow some exploration path in  $\mathcal{E}$ , the *search path*, until the goal has been seen. We assume that all search and exploration paths return to the start point,  $s$ , and we make the usual assumption that goals must be at least a distance 1 away from the start point.<sup>1</sup>

For  $d \geq 1$ , let  $\mathcal{E}(d)$  denote the part of  $\mathcal{E}$  in distance at most  $d$  from  $s$ . A *depth- $d$  restricted exploration* explores all potential goal positions in  $\mathcal{G}(d)$ . The exploration path may move outside  $\mathcal{E}(d)$  as long as it stays within  $\mathcal{E}$ . Depth- $d$  restricted search is defined accordingly.

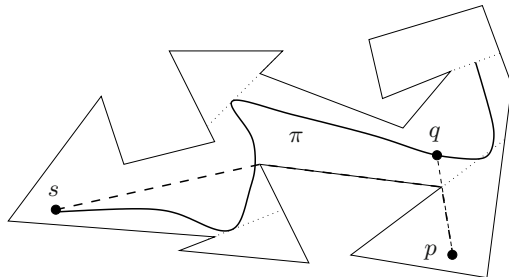
It remains to define what it means that the searcher “sees” the goal. In graph searching, agents are usually assumed to be *blind*, i. e., standing at a vertex of a directed graph the agent sees the set of outgoing edges, but neither their lengths nor the position of the other vertices are known. Incoming edges cannot be sensed; see [7]. Blind agents must eventually visit all points in the goal set.

Polygon searchers are usually assumed to have *vision*, that is, they always know the current visibility polygon. Such agents need not visit a goal position if they can see it from somewhere else. Searching a polygon means to visit all visibility cuts, i. e., all rays extending the edges of the reflex vertices. Actually, there is always a subset of the cuts, the *essential cuts*, whose visit guarantees that all other cuts are also visited on the way.

In case of online algorithms, we assume that agents have perfect memory. They always know a map of the part of  $\mathcal{E}$  already explored, and they can always recognize when they visit some point for the second time, i. e., they have perfect localization (the robot localization problem is actually a difficult problem by itself, see for example [10]).

<sup>1</sup> If goals could be arbitrarily close to  $s$ , no algorithm could be competitive.

We now introduce a few notations. Let  $\pi$  be a path in the environment  $\mathcal{E}$ . For a point  $p \in \pi$  let  $\pi(p)$  denote the part of  $\pi$  between  $s$  and  $p$ , and  $\text{sp}(p)$  the shortest path from  $s$  to  $p$  in  $\mathcal{E}$ . We denote the length of a path segment  $\pi(p)$  by  $|\pi(p)|$ . Paths computed by some algorithm  $\mathcal{A}$  will be named  $\mathcal{A}$ , too. For a point  $p \in \mathcal{E}$  let  $\text{rise}_\pi(p)$  denote the point  $q \in \pi$  from which  $p$  is seen for the first time when moving along  $\pi$  starting at  $s$ , see Fig. 1.



**Fig. 1.** A search path  $\pi$  in a polygon, visiting all essential cuts (the dotted lines). The dashed path is the shortest path  $\text{sp}(p)$  from  $s$  to the goal  $p$ . Moving along  $\pi$ ,  $p$  can first be seen from  $q = \text{rise}_\pi(p)$ .

The quality of a search path is measured by its *search ratio*  $\text{sr}(\pi)$ , defined as  $\text{sr}(\pi) := \max_{p \in \mathcal{G}} \frac{|\pi(q)| + |qp|}{|\text{sp}(p)|}$ , where  $q = \text{rise}_\pi(p)$ . Note that  $q = p$  for blind agents. An *optimal search path*,  $\pi_{\text{opt}}$ , is a search path with a minimum search ratio  $\text{sr}_{\text{opt}} = \text{sr}(\pi_{\text{opt}})$ .

Since the optimal search path seems hard to compute [14], we are interested in finding good approximations of the optimal search path, in offline and online scenarios. We say a search path  $\pi$  is *C-search-competitive* (with respect to the optimal search path  $\pi_{\text{opt}}$ ) if  $\text{sr}(\pi) \leq C \cdot \text{sr}(\pi_{\text{opt}})$ . Note that  $\pi$  is then a  $C \cdot \text{sr}(\pi_{\text{opt}})$ -competitive search algorithm (in the usual competitive sense).

### 3 A General Approximation Framework

In this section we will show how to transform an exploration algorithm, offline or online, into a search algorithm, without losing too much on the approximation factor.

Let  $\mathcal{E}$  be the given environment and  $\pi_{\text{opt}}$  an optimal search path. We assume that, for any point  $p$ , we can reach  $s$  from  $p$  on a path of length at most  $\text{sp}(p)$ .<sup>2</sup>

For  $d \geq 1$ , let  $\text{Expl}(d)$  be a family of depth- $d$  restricted exploration algorithms for  $\mathcal{E}$ , either online or offline. Let  $\text{OPT}$  and  $\text{OPT}(d)$  denote the corresponding optimal offline depth- $d$  restricted exploration algorithms.

<sup>2</sup> Note that this is not the case for directed graphs, but it holds for undirected graphs and polygonal environments. We will see later that there is no constant-competitive online search algorithm for directed graphs, anyway.

**Definition 1.** The family  $\text{Expl}(d)$  is *DREP (depth restricted exploration property)* if there are constants  $\beta > 0$  and  $C_\beta \geq 1$  such that, for any  $d \geq 1$ ,  $\text{Expl}(d)$  is  $C_\beta$ -competitive against the optimal algorithm  $\text{OPT}(\beta d)$ , i.e.,  $|\text{Expl}(d)| \leq C_\beta \cdot |\text{OPT}(\beta d)|$ .  $\square$

In the normal competitive framework we would compare  $\text{Expl}(d)$  to the optimal algorithm  $\text{OPT}(d)$ , i.e.,  $\beta = 1$ . As we will see later, our more general definition makes it sometimes easier to find DREP exploration algorithms. Usually, we cannot just take an exploration algorithm  $\text{Expl}$  for  $\mathcal{E}$  and restrict it to points in distance at most  $d$  from  $s$ . This way, we might miss useful shortcuts outside of  $\mathcal{E}(d)$ . Even worse, it may not be possible to determine in an online setting which parts of the environment belong to  $\mathcal{E}(d)$ , making it difficult to explore the right part of  $\mathcal{E}$ . In the next two sections we will derive DREP exploration algorithms for graphs and polygons by carefully adapting existing exploration algorithms for the entire environment.

To obtain a search algorithm for  $\mathcal{E}$  we use the *doubling strategy*. For  $i = 1, 2, 3, \dots$ , we successively run the exploration algorithm  $\text{Expl}(2^i)$ , each time starting at  $s$ .

**Theorem 1.** *The doubling strategy based on a DREP exploration strategy is a  $4\beta C_\beta$ -search-competitive (plus an additive constant) search algorithm for blind agents, and a  $8\beta C_\beta$ -search-competitive (plus an additive constant) search algorithm for agents with vision.*

*Proof.* Consider one iteration of the doubling strategy with search radius  $d \geq 1$ . Let  $\text{last}(d)$  be the point on the optimal search path  $\pi_{\text{opt}}$  for  $\mathcal{E}$  from which we see the last point in distance at most  $d$  from  $s$  when moving along  $\pi_{\text{opt}}$ . If the agent has vision,  $\text{last}(d)$  could lie outside of  $\mathcal{E}(d)$ . Note that  $|\text{OPT}(d)| \leq |\pi_{\text{opt}}(\text{last}(d))| + |\text{sp}(\text{last}(d))|$ . For a blind agent we have  $\text{sp}(\text{last}(d)) \leq d$ . Thus,  $\text{sr}_{\text{opt}} \geq \frac{|\pi_{\text{opt}}(\text{last}(d))|}{d} \geq \frac{|\text{OPT}(d)| - d}{d}$ , or  $|\text{OPT}(d)| \leq d \cdot (\text{sr}_{\text{opt}} + 1)$ . If the goal is in distance  $2^j + \epsilon$  for some small  $\epsilon > 0$ , then the search ratio of the doubling strategy is bounded by  $\frac{\sum_{i=1}^{j+1} |\text{Expl}(2^i)|}{2^j} \leq C_\beta \cdot \frac{\sum_{i=1}^{j+1} |\text{OPT}(\beta 2^i)|}{2^j} \leq C_\beta \cdot \frac{\sum_{i=1}^{j+1} \beta 2^i \cdot (\text{sr}_{\text{opt}} + 1)}{2^j} \leq 4\beta C_\beta \cdot (\text{sr}_{\text{opt}} + 1)$ .

If the agent has vision, we only know that  $|\text{sp}(\text{last}(d))| \leq |\pi_{\text{opt}}(\text{last}(d))|$ . Thus,  $\text{sr}_{\text{opt}} \geq \frac{|\pi_{\text{opt}}(\text{last}(d))|}{d} \geq \frac{|\text{OPT}(d)|}{2d}$ , or  $|\text{OPT}(d)| \leq 2d \cdot \text{sr}_{\text{opt}}$ . So in this case we can bound the search ratio by  $\frac{2^j + \sum_{i=1}^{j+1} |\text{Expl}(2^i)|}{2^j} \leq 1 + 8\beta C_\beta \cdot \text{sr}_{\text{opt}}$ .  $\square$

The only problem is now to find good DREP exploration algorithms for various environments.

## 4 Searching Graphs

We distinguish between graphs with unit length vs. arbitrary length edges, planar vs. non-planar graphs, directed vs. undirected graphs, and vertex vs. geometric search. We assume agents are blind, i.e., they can at any vertex only

sense the number of outgoing edges but they cannot sense the incoming edges and the endpoints of the outgoing edges. In the vertex search problem, we assume w.l.o.g. that graphs do not have parallel edges. Otherwise, there can be no constant-search-competitive vertex search algorithm. In Fig. 2(iv), the optimal search path  $s \rightarrow v \rightarrow t \rightarrow s$  has length 3, whereas any online search path can be forced to cycle often between  $s$  and  $v$  before traversing the edge  $v \rightarrow t$ . Note that we also could use undirected edges.

#### 4.1 Non-competitiveness Results

We first show that for many variants there is no constant-search-competitive online search algorithm. Incidentally, there is also no constant-competitive online exploration algorithm for these graph classes. Note that non-search-competitiveness for planar graphs implies non-search-competitiveness for general graphs, non-search-competitiveness for unit length edges implies non-search-competitiveness for arbitrary length edges, and non-search-competitiveness for undirected graphs implies non-search-competitiveness for directed graphs (we could replace each undirected edge with directed edges in both directions).

**Theorem 2.** *For blind agents, there is no constant-search-competitive online vertex search algorithm for (i) non-planar graphs, (ii) directed planar graphs, (iii) planar graphs with arbitrary edge lengths. Further, there is no constant-search-competitive online geometric search algorithm for directed graphs with unit length edges.*

*Proof.* It is not difficult to verify the claims on the graphs in Fig. 2(i)-(iii).  $\square$

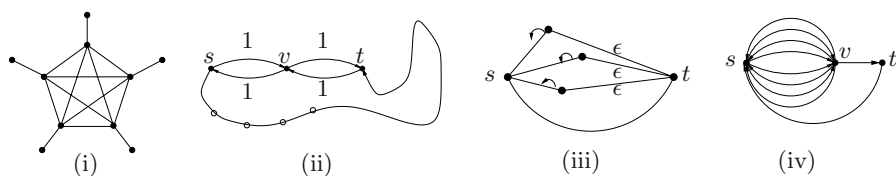


Fig. 2. Lower bound constructions of Theorem 2.

#### 4.2 Competitive Search in Graphs

In this subsection we will present search-competitive online and offline search algorithms for the remaining graph classes. We assume in this subsection that graphs are always undirected.

**Trees.** On trees, *DFS* is a 1-competitive online exploration algorithm for vertex and geometric search that is DREP; it is still 1-competitive when restricted to search depth  $d$ , for any  $d \geq 1$ . Thus, the doubling strategy gives a polynomial time 4-search-competitive search algorithm for trees, online and offline. On the other hand, it is an open problem whether the computation of an optimal vertex or geometric search path in trees with unit length edges is NP-complete [14].

**Competitive Graph Search Algorithms.** We will now give competitive search algorithms for planar graphs with unit length edges (vertex search) and for general graphs with arbitrary length edges (geometric search). Both algorithms are based on an online algorithm for online tethered graph exploration.

In the *tethered exploration* problem the agent is fixed to the start point by a restricted length rope. An optimal solution to this problem was given by Duncan et al. [9]. Their algorithm can explore an unknown graph with unit length edges in  $2|E| + (4 + \frac{16}{\alpha})|V|$  edge traversals, using a rope length of  $(1 + \alpha)d$ , where  $d$  is the distance of the point farthest away from the start point and  $\alpha > 0$  is some parameter. As they pointed out, the algorithm can also be used for depth restricted exploration. To explore all edges in  $G((1 + \alpha)d)$ , for  $d \geq 1$ , their algorithm uses at most  $2|E((1 + \alpha)d)| + (4 + \frac{16}{\alpha})|V((1 + \alpha)d)|$  edge traversals using a rope of length  $(1 + \alpha)d$ . Let us call this algorithm *Expl*( $d$ ). The algorithm explores the graph in a mixture of bounded-depth *DFS* on  $G$ , *DFS* on some spanning tree of  $G$ , and recursive calls to explore certain large subgraphs. The bound for the number of edge traversals can intuitively be explained as follows: bounded-depth *DFS* visits each edge at most twice, thus the term  $2|E((1 + \alpha)d)|$ ; *DFS* on the spanning tree visits every node at most twice, but nodes can be in two overlapping spanning trees, thus the term  $4|V((1 + \alpha)d)|$ ; relocating between recursive calls does not happen too often (because of the size of the subgraphs), giving the term  $\frac{16}{\alpha}|V((1 + \alpha)d)|$ .

We note that *Expl*( $d$ ) can be modified to run on graphs with arbitrary length edges. Then we do not bound the number of edge traversals, but the total length of all traversed edges. Let  $length(E)$  denote the total length of all edges in  $E$ . It is possible to adapt the proofs in [9] to prove the following lemma.

**Lemma 1.** *In graphs with arbitrary length edges,  $Expl(d)$  explores all edges and vertices in  $G(d)$  using a rope of length  $(1 + \alpha)d$  at a cost of at most  $(4 + \frac{8}{\alpha}) \cdot length(E((1 + \alpha)d))$ .*

*Proof. (Sketch)* Intuitively, *DFS* and bounded-depth *DFS* traverse each edge at most twice, thus the term  $4 \cdot length(E((1 + \alpha)d))$ ; relocating between recursive calls does not happen too often and subgraphs do not overlap (at least not their edges), thus the term  $\frac{8}{\alpha} \cdot length(E((1 + \alpha)d))$ .  $\square$

**Lemma 2.** *In planar graphs with unit length edges,  $Expl(d)$  is a DREP online vertex exploration algorithm with  $\beta = 1 + \alpha$  and  $C_\beta = 10 + \frac{16}{\alpha}$ .*

*Proof.*  $E((1 + \alpha)d) \leq 3V((1 + \alpha)d) - 6$  by Euler's formula. Thus, the number of edge traversals is at most  $(10 + \frac{16}{\alpha})V((1 + \alpha)d)$ . On the other hand,  $OPT((1 + \alpha)d)$  must visit each vertex in  $V((1 + \alpha)d)$  at least once.  $\square$

**Theorem 3.** *The doubling strategy based on  $\text{Expl}(d)$  is a  $(104 + 40\alpha + \frac{64}{\alpha})$ -search-competitive online vertex search algorithm for blind agents in planar graphs with unit length edges. The competitive ratio is minimal for  $\alpha = \sqrt{\frac{16}{10}}$ .  $\square$*

**Lemma 3.** *In general graphs with arbitrary length edges,  $\text{Expl}(d)$  is a DREP online geometric exploration algorithm with  $\beta = 1 + \alpha$  and  $C_\beta = 4 + \frac{8}{\alpha}$ .*

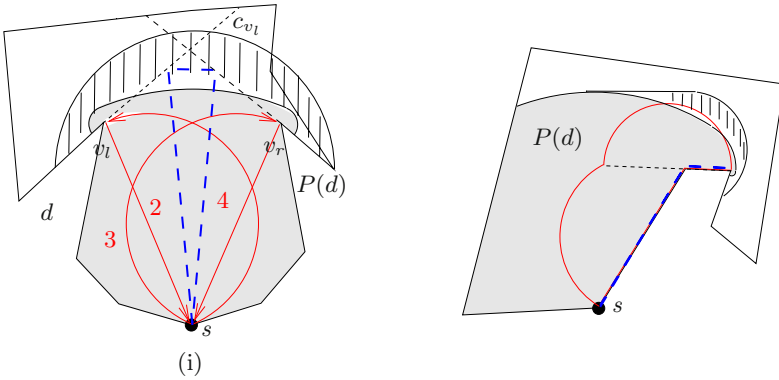
*Proof.* The total cost of  $\text{Expl}(d)$  is at most  $(4 + \frac{8}{\alpha}) \cdot \text{length}(E((1 + \alpha)d))$  by Lemma 1. On the other hand,  $\text{OPT}((1 + \alpha)d)$  must traverse each edge in  $E((1 + \alpha)d)$  at least once.  $\square$

**Theorem 4.** *The doubling strategy based on  $\text{Expl}(d)$  is a  $(48 + 16\alpha + \frac{32}{\alpha})$ -search-competitiveonline geometric search algorithm for blind agents in general graphs with arbitrary length edges.  $\square$*

## 5 Searching Polygons

### 5.1 Simple Polygons

A simple polygon  $P$  is given by a closed non-intersecting polygonal chain. We assume that agents have vision. To apply our framework we need a DREP online exploration algorithm  $\text{Expl}_{\text{onl}}(d)$ .



**Fig. 3.** (i)  $PE(d)$  explores the left reflex vertex  $v_l$  along a circular arc (1), returns to the start (2) and explores the right reflex vertex  $v_r$  likewise (3)+(4). On the other hand, the shortest exploration path for  $P(d)$  in  $P$ , the dashed line, leaves  $P(d)$ . But we can extend  $P(d)$  by the circular arc. (ii)  $PE(d)$  leaves  $P(d)$  whereas the shortest exploration path for  $P(d)$  lies inside  $P(d)$ .

The only known algorithm,  $PE$ , for the online exploration of a simple polygon by Hoffmann et al. [11] achieves a competitive ratio of 26.5. Now we must adapt



this algorithm to depth restricted exploration. The undetected parts of  $P$  always lie behind cuts  $c_v$  emanating from reflex vertices  $v$ . These reflex vertices are called *unexplored* as long as we have not visited the corresponding cut  $c_v$ . We modify  $PE$  so that it explores  $P$  only up to distance  $d$  from  $s$ . The algorithm always maintains a list of unexplored reflex vertices and successively visits the corresponding cuts. While exploring a reflex vertex (along a sequence of line segments and circular arcs), more unexplored reflex vertices may be detected or unexplored reflex vertices may become explored. These vertices are inserted into or deleted from the list, respectively. In  $PE(d)$ , unexplored reflex vertices in a distance greater than  $d$  from the start will be ignored, i.e., although they may be detected they will not be inserted into the list. Let  $\text{OPT}(P, d)$  be the shortest path that sees all points in  $P(d)$ .

Note that  $\text{OPT}(P, d)$  and  $PE(d)$  may leave  $P(d)$ , see Fig. 3. Nevertheless, it is possible to adapt the analysis of Hoffmann et al. There are actually two ways to do this. Either we enlarge  $P(d)$  without creating new reflex vertices such that the larger polygon contains  $\text{OPT}(P, d)$  and  $PE(d)$ . Or we redo the analysis of Hoffmann et al. in  $P$  restricted to  $\text{OPT}(P, d)$  and  $PE(d)$ , which is possible. In the former case we can use some sort of extended boundary around the boundary of  $P(d)$  such that the enlarged polygon contains  $PE(d)$  and  $\text{OPT}(P, d)$ , see the convex enlargements in Fig. 3. It may happen that these new extensions overlap for different parts of  $P(d)$ . However, this does not affect the analysis of Hoffmann et al.

**Lemma 4.** *In a simple polygon,  $PE(d)$  is a DREP online exploration algorithm with  $\beta = 1$  and  $C_\beta = 26.5$ .  $\square$*

**Theorem 5.** *The doubling strategy based on  $PE(d)$  is a 212-search-competitive online search algorithm for an agent with vision in a simple polygon. There is also a polynomial time 8-search-competitive offline search algorithm.*

*Proof.* The online search-competitiveness follows from Lemma 4 and Theorem 1.

If we know the polygon, we can compute  $\text{OPT}(P, d)$  in polynomial time by adapting a corresponding algorithm for  $P$ . Every known polynomial time offline exploration algorithm builds a sequence of the essential cuts, see for example [5, 18, 17, 8]. Any of these algorithms could be used in our framework. Since an optimal algorithm has approximation factor  $C = 1$ , our framework yields an approximation of the optimal search ratio within a factor of 8.

If we skip a step with distance  $2^i$  if there is no reflex vertex within a distance between  $2^{i-1}$  and  $2^i$ , the total running time is bounded by a polynomial in the number of the vertices of  $P$ .  $\square$

Now the question arises whether there is a polynomial time algorithm that computes the optimal search path in a simple polygon. All essential cuts need to be visited, so we can try to visit them in any possible order. However, we do not know where exactly we should visit a cut. We are not sure whether there are only a few possibilities (similar to the shortest watchman route problem), i.e.,

whether this subproblem is discrete. So the problem of efficiently computing an optimal search path in a polygon is still open.

For rectilinear simple polygons we can find better online algorithms based on a  $\sqrt{2}$ -competitive online exploration algorithm by Papadimitriou et al. [6] which can be made DREP by ignoring reflex vertices farther away than  $d$ . Again, no polynomial time algorithm for the optimal search path is known.

**Theorem 6.** *For an agent with vision in a simple rectilinear polygon there is a  $8\sqrt{2}$ -search-competitive online search algorithm. There is also a polynomial time 8-search-competitive offline search algorithm.*  $\square$

## 5.2 Polygons with Holes

We will show that there is no constant-search-competitive online search algorithm for polygons with rectangular holes. It was shown by Albers et al. [1] that there is no constant-competitive online exploration algorithm for polygons with rectangular holes. For  $k \geq 2$ , they filled a rectangle of height  $k$  and width  $2k$  with  $O(k^2)$  rectangular holes such that the optimal exploration tour has length  $O(k)$ , whereas any online exploration algorithm needs to travel a distance of  $\Omega(k^2)$ . The details of the construction are not important here. We just note that it has the property that any point  $p$  is at most at distance  $3k$  from the start point, which is in the lower left corner of the bounding rectangle.

**Theorem 7.** *For an agent with vision in a polygon with rectangular holes there is no constant-search-competitive online search algorithm.*

*Proof.* We extend the construction of Albers et al. by making the bounding rectangle larger and placing a new hole of height  $k$  and width  $2k$  just below all the holes of the previous construction. The new start point  $s$  is again in the lower left corner of the bounding rectangle. Any point that is not immediately visible from  $s$  has at least distance  $k$  from  $s$ .  $\square$

Since the offline exploration problem is NP-complete (by straightforward reduction from planar TSP) we cannot use our framework to obtain a polynomial time approximation algorithm of the optimal search path. However, there is an exponential time 8-approximation algorithm. We can list the essential cuts of  $\text{OPT}(P, d)$  in any order to find the best one. Application of our framework then gives an approximation factor of 8 for the optimal search ratio.

The results of Koutsoupias et al. [14] imply that the offline problem of computing an optimal search path in a known polygon with (rectangular) holes is NP-complete.

## 6 Conclusion and Open Problems

We have introduced a framework for computing online and offline approximations of the optimal search path graph and polygonal environments. We have

obtained fairly simple proofs of the existence of approximation strategies and their approximation factors, although the factors are quite high. We have also shown that some environments do not have constant-search-competitive online search strategies. Our framework would also work for randomized algorithms, but there are not many randomized exploration algorithms around. For some of the settings it remains open whether the offline optimization problem is NP-hard.

## References

1. Susanne Albers, Klaus Kursawe, and Sven Schuierer. Exploring unknown environments with obstacles. In *Proc. 10th SODA*, pp. 842–843, 1999.
2. Steve Alpern and Shmuel Gal. *The Theory of Search Games and Rendezvous*. Kluwer Academic Publications, 2002.
3. R. Baeza-Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Inform. Comput.*, 106:234–252, 1993.
4. Piotr Berman. On-line searching and navigation. In A. Fiat and G. Woeginger, eds., *Competitive Analysis of Algorithms*. Springer, 1998.
5. W.-P. Chin and S. Ntafos. Shortest watchman routes in simple polygons. *Discrete Comput. Geom.*, 6(1):9–31, 1991.
6. Xiaotie Deng, Tiko Kameda, and Christos Papadimitriou. How to learn an unknown environment I: The rectilinear case. *Journal of the ACM*, 45(2):215–245, 1998.
7. Xiaotie Deng and Christos H. Papadimitriou. Exploring an unknown graph. *Journal of Graph Theory*, 32:265–297, 1999.
8. M. Dror, A. Efrat, A. Lubiwi, and J. S. B. Mitchell. Touring a sequence of polygons. In *Proc. 35th STOC*, pp. 473–482, 2003.
9. Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal constrained graph exploration. In *Proc. 12th SODA*, pp. 307–314, 2001.
10. Rudolf Fleischer, Kathleen Romanik, Sven Schuierer, and Gerhard Trippen. Optimal robot localization in trees. *Information and Computation*, 171:224–247, 2001.
11. Frank Hoffmann, Christian Icking, Rolf Klein, and Klaus Kriegel. The polygon exploration problem. *SIAM Journal on Computing*, 31:577–600, 2001.
12. Christian Icking, Rolf Klein, and Elmar Langetepe. An optimal competitive strategy for walking in streets. In *Proc. 16th STACS*, LNCS 1563, pp. 110–120. Springer, 1999.
13. R. Klein. *Algorithmische Geometrie*. Addison-Wesley Longman, 1997.
14. Elias Koutsoupias, Christos H. Papadimitriou, and Mihalis Yannakakis. Searching a fixed graph. In *Proc. 23th ICALP*, LNCS 1099, pp. 280–289. Springer, 1996.
15. S. Schuierer. On-line searching in simple polygons. In H. Christensen, H. Bunke, and H. Noltemeier, eds., *Sensor Based Intelligent Robots*, LNAI 1724, pp. 220–239. Springer, 1997.
16. Sven Schuierer and Ines Semrau. An optimal strategy for searching in unknown streets. In *Proc. 16th STACS*, LNCS 1563, pp. 121–131. Springer, 1999.
17. X. Tan, T. Hirata, and Y. Inagaki. Corrigendum to “an incremental algorithm for constructing shortest watchman routes”. *Internat. J. Comput. Geom. Appl.*, 9(3):319–323, 1999.
18. X. H. Tan, T. Hirata, and Y. Inagaki. An incremental algorithm for constructing shortest watchman routes. *Internat. J. Comput. Geom. Appl.*, 3(4):351–365, 1993.

# Incremental Algorithms for Facility Location and $k$ -Median<sup>\*</sup>

Dimitris Fotakis

Department of Mathematical, Physical and Computational Sciences,  
Aristotle University of Thessaloniki, 54006 Thessaloniki, Greece.  
Computer Technology Institute, Patras, Greece.  
fotakis@auth.gr

**Abstract.** In the incremental versions of Facility Location and  $k$ -Median, the demand points arrive one at a time and the algorithm must maintain a good solution by either adding each new demand to an existing cluster or placing it in a new singleton cluster. The algorithm can also merge some of the existing clusters at any point in time. We present the first incremental algorithm for Facility Location with uniform facility costs which achieves a constant performance ratio and the first incremental algorithm for  $k$ -Median which achieves a constant performance ratio using  $O(k)$  medians.

## 1 Introduction

The model of incremental algorithms for data clustering is motivated by practical applications where the demand sequence is not known in advance and the algorithm must maintain a good clustering using a restricted set of operations which result in a solution of hierarchical structure. Charikar et al. [3] introduced the framework of incremental clustering to meet the requirements of data classification applications in information retrieval.

In this paper, we consider the incremental versions of metric Facility Location and  $k$ -Median. In *Incremental  $k$ -Median* [5], the demand points arrive one at a time. Each demand must be either added to an existing cluster or placed in a new singleton cluster upon arrival. At any point in time, the algorithm can also merge some of the existing clusters. Each cluster is represented by its median whose location is determined at the cluster's creation time. When some clusters are merged with each other, the median of the new cluster must be selected among the medians of its components. The goal is to maintain a solution consisting of at most  $k$  clusters/medians which minimize the total assignment cost of the demands considered so far. The assignment cost of a demand is its distance from the median of the cluster the demand is currently included in.

---

<sup>\*</sup> This work was partially supported by the FET programme of the EU under contract numbers IST-1999-14186 (ALCOM-FT) and IST-2001-33116 (FLAGS). Part of this work was done while the author was at the Max-Planck-Institut für Informatik, Saarbrücken, Germany.

In *Incremental Facility Location*, demand points arrive one at a time and must be assigned to either an existing or a new facility upon arrival. At any point in time, the algorithm can also merge a facility with another one by closing the first facility and re-assigning all the demands currently assigned to it to the second facility. The objective is to maintain a solution which minimizes the sum of facility and assignment costs. The facility cost only accounts for the facilities currently open, while the assignment cost of a demand is its distance from the facility the demand is currently assigned to.

Both problems are motivated by applications of clustering in settings that the demands arrive online (e.g., clustering the web, data mining) and the algorithm must maintain a good solution without resorting to complete re-clustering. To avoid cluster proliferation, the algorithm is allowed to merge existing clusters. In Incremental  $k$ -Median, the number of clusters is fixed in advance. On the other hand, Incremental Facility Location provides a model for applications where the number of clusters cannot be determined in advance due to limited a priori information about the demand sequence. Hence, we introduce a uniform facility cost enforcing the requirement for a relatively small number of clusters.

We evaluate the performance of incremental algorithms using the *performance ratio* [3], whose definition is essentially identical to that of the *competitive ratio* (e.g., [2]). An incremental algorithm achieves a performance ratio of  $c$  if for all demand sequences, the algorithm's cost is at most  $c$  times the cost of an optimal offline algorithm, which has full knowledge of the demand sequence, on the same instance. We also let  $n$  denote the total number of demands.

**Comparison to Online and Streaming Algorithms.** Similarly to online algorithms, incremental algorithms commit themselves to irrevocable decisions made without any knowledge about future demands. More specifically, when a new demand arrives, the algorithm may decide to add the demand to an existing cluster or merge some clusters with each other. These decisions are irrevocable because once formed, clusters cannot be broken up. However, we are not aware of any simple and natural notion of irrevocable cost that could be associated with the decision that some demands are clustered together.

Incremental  $k$ -Median can be regarded as the natural online version of  $k$ -Median. Nevertheless, Incremental Facility Location is quite different from the problem of Online Facility Location (OFL) introduced in [10]. In particular, OFL is motivated by network design applications where re-configurations are expensive and often infeasible. Thus, the decisions of opening a facility at a particular location and assigning a demand to a facility are irrevocable as well as the corresponding costs. On the other hand, Incremental Facility Location is motivated by clustering applications where merging existing clusters is not only feasible but also desirable and the important restriction is that existing clusters should not be broken up. Hence, only the decision that some demands are clustered together is irrevocable. As a result, the competitive ratio is  $\Theta(\frac{\log n}{\log \log n})$  for OFL [6] and constant for Incremental Facility Location.

Incremental algorithms also bear a resemblance to one-pass streaming algorithms for clustering problems (e.g., [9] for a formal definition of the streaming

model of computation). In case of streaming algorithms, the emphasis is on space and time efficient algorithms which achieve a small approximation ratio by ideally performing a single scan over the input. A streaming algorithm for  $k$ -Median is not restricted in terms of the solution's structure or the set of operations available. On the other hand, incremental algorithms must maintain a good hierarchical clustering by making irrevocable decisions, but they should only run in polynomial time. Nevertheless, all known incremental algorithms for clustering problems can be either directly regarded as or easily transformed to efficient one-pass streaming algorithms (e.g., [3,8,5,4] and this paper).

**Previous Work.** Charikar et al. [3] presented incremental algorithms for  $k$ -Center which achieve a constant performance ratio using  $k$  clusters. Charikar and Panigrahy [5] presented an incremental algorithm for Sum  $k$ -Radius which achieves a constant performance ratio using  $O(k)$  clusters. They also proved that any deterministic incremental algorithm for  $k$ -Median which maintains at most  $k$  clusters must have a performance ratio of  $\Omega(k)$ . Determining whether there exists an incremental algorithm which achieves a constant performance ratio using  $O(k)$  medians is suggested as an open problem in [5].

In [8,4], they are presented one-pass streaming algorithms for  $k$ -Median which assume that  $n$  is known in advance. For  $k$  much smaller than  $n^\epsilon$ , the algorithms of Guha et al. [8] achieve a performance ratio of  $2^{O(1/\epsilon)}$  using  $n^\epsilon$  medians and run in  $O(nk \text{ poly}(\log n))$  time and  $n^\epsilon$  space. The algorithm of Charikar et al. [4] achieves a constant performance ratio with high probability (whp.) using  $O(k \log^2 n)$  medians and runs in  $O(nk \log^2 n)$  time and  $O(k \log^2 n)$  space.

The algorithms of [10,6,1] for OFL are the only known incremental algorithms for Facility Location. In OFL [10], the demands arrive one at a time and must be irrevocably assigned to either an existing or a new facility upon arrival. In [10], a randomized  $O(\frac{\log n}{\log \log n})$ -competitive algorithm and a lower bound of  $\omega(1)$  are presented. In [6], the lower bound is improved to  $\Omega(\frac{\log n}{\log \log n})$  and a deterministic  $O(\frac{\log n}{\log \log n})$ -competitive algorithm is given. In [1], it is presented a simpler and faster deterministic  $O(2^d \log n)$ -competitive algorithm for  $d$ -dimensional Euclidean spaces.

The lower bounds of [10,6] hold only if the decision of opening a facility at a particular location is irrevocable. Hence, they do not apply to Incremental Facility Location. However, the lower bound of [6] implies that every algorithm which maintains  $o(k \log n)$  facilities must incur a total *initial* assignment cost of  $\omega(1)$  times the optimal cost, where the initial assignment cost of a demand is its distance from the first facility the demand is assigned to. Therefore, every algorithm treating merge as a black-box operation cannot approximate the optimal assignment cost within a constant factor unless it uses  $\Omega(k \log n)$  facilities (e.g., the algorithm of [4]). In other words, to establish a constant performance ratio, one needs a merge rule which can provably *decrease* both the algorithm's facility and assignment costs.

**Contribution.** We present the first incremental algorithm for metric Facility Location with uniform facility costs which achieves a constant performance ra-

tio. The algorithm combines a simple rule for opening new facilities with a novel merge rule based on distance instead of cost considerations. We use a new technique to prove that a case similar to the special case that the optimal solution consists of a single facility is the dominating case in the analysis. This technique is also implicit in [6] and may find applications to other online problems. To overcome the limitation imposed by the lower bound of [6], we establish that in the dominating case, merge operations also decrease the assignment cost.

Using the algorithm for Facility Location as a building block, we obtain the first incremental algorithm for  $k$ -Median which achieves a constant performance ratio using  $O(k)$  medians, thus resolving the open question of [5]. In other words, we improve the number of medians required for a constant performance ratio from  $O(k \log^2 n)$  to  $O(k)$ . Combining our techniques with the techniques of [4], we obtain a randomized incremental algorithm which achieves a constant performance ratio whp. using  $O(k)$  medians and runs in  $O(nk^2 \log^2 n)$  time and  $O(k^2 \log^2 n)$  space. This algorithm can also be regarded as a time and space efficient one-pass streaming algorithm for  $k$ -Median.

The proofs and the technical details omitted from this extended abstract due to space constraints can be found in the full version of this paper [7].

**Notation.** We only consider unit demands and allow multiple demands to be located at the same point. For Incremental Facility Location, we restrict our attention to the special case of uniform facility costs, where the cost of opening a facility, denoted by  $f$ , is the same for all points. We also use the terms facility, median and cluster interchangeably.

A metric space  $\mathcal{M} = (M, d)$  is usually identified by its point set  $M$ . The distance function  $d$  is non-negative, symmetric, and satisfies the triangle inequality. For points  $u, v \in M$  and a subspace  $M' \subseteq M$ ,  $d(u, v)$  denotes the distance between  $u$  and  $v$ ,  $D(M')$  denotes the diameter of  $M'$ , and  $d(M', u)$  denotes the distance between  $u$  and the nearest point in  $M'$ . It is  $d(\emptyset, u) = \infty$ . For a point  $u \in M$  and a non-negative number  $r$ ,  $\text{Ball}(u, r) \equiv \{v \in M : d(u, v) \leq r\}$ .

## 2 An Incremental Algorithm for Facility Location

The algorithm Incremental Facility Location - IFL (Fig. 1) maintains its *facility configuration*  $F$ , its *merge configuration* consisting of a *merge ball*  $\text{Ball}(w, m(w))$  for each facility  $w \in F$ , and the set  $L$  of *unsatisfied demands*.

A demand becomes unsatisfied, i.e., is added to the set  $L$ , upon arrival. Each unsatisfied demand holds a *potential* always equal to its distance to the nearest existing facility. If the unsatisfied neighborhood  $B_u$  of a new demand  $u$  has accumulated a potential of  $\beta f$ , a new facility located at the same point with  $u$  opens. Then, the demands in  $B_u$  become satisfied and lose their potential. Hence, the notion of unsatisfied demands ensures that each demand contributes to the facility cost at most once.

Each facility  $w \in F$  maintains the set  $\text{Init}(w)$  of the demands *initially assigned* to  $w$  (namely, the demands assigned to  $w$  upon arrival and not after a merge operation), its *merge radius*  $m(w)$  and the corresponding *merge ball*



<p>Let <math>x</math>, <math>\beta</math>, and <math>\psi</math> be constants.  <math>F \leftarrow \emptyset</math>; <math>L \leftarrow \emptyset</math>;          For each new demand <math>u</math>:  <math>L \leftarrow L \cup \{u\}</math>; <math>r_u \leftarrow d(F, u)/x</math>;  <math>B_u \leftarrow \text{Ball}(u, r_u) \cap L</math>;  <math>\text{Pot}(B_u) = \sum_{v \in B_u} d(F, v)</math>;          if <math>\text{Pot}(B_u) \geq \beta f</math> then              Let <math>w'</math> be the location of <math>u</math>;              <math>\text{open}(w')</math>; <math>L \leftarrow L \setminus B_u</math>;              for each <math>w \in F \setminus \{w'\}</math> do                  if <math>d(w, w') \leq m(w)</math> then                      <math>\text{merge}(w \rightarrow w')</math>;              Let <math>w</math> be the facility in <math>F</math> closest to <math>u</math>;              <math>\text{update\_merge\_radius}(m(w))</math>;              <math>\text{initial\_assignment}(u, w)</math>;</p>	<p><math>\text{open}(w')</math>  <math>F \leftarrow F \cup \{w'\}</math>; <math>\text{Init}(w') \leftarrow \emptyset</math>;  <math>C(w') \leftarrow \emptyset</math>; <math>m_1(w') \leftarrow 3r_u</math>;  <math>\text{merge}(w \rightarrow w')</math>  <math>F \leftarrow F \setminus \{w\}</math>;  <math>C(w') \leftarrow C(w') \cup C(w)</math>;  <math>\text{update\_merge\_radius}(m(w))</math>  <math>m_2(w) = \max\{r :</math>  <math> \text{Ball}(w, \frac{r}{\psi}) \cap (\text{Init}(w) \cup \{u\})  \cdot r \leq \beta f\}</math>;  <math>m(w) = \min\{m_1(w), m_2(w)\}</math>;  <math>\text{initial\_assignment}(u, w)</math>  <math>\text{Init}(w) \leftarrow \text{Init}(w) \cup \{u\}</math>;  <math>C(w) \leftarrow C(w) \cup \{u\}</math>;</p>
---	---

**Fig. 1.** The algorithm Incremental Facility Location – IFL .

$\text{Ball}(w, m(w))$ .  $w$  is the only facility in its merge ball. In particular, when  $w$  opens, the merge radius of  $w$  is initialized to a fraction of the distance between  $w$  and the nearest existing facility. Then, if a new facility  $w'$  opens in  $w$ 's merge ball,  $w$  is merged with  $w'$ . The algorithm keeps decreasing  $m(w)$  to ensure that no merge operation can dramatically increase the assignment cost of the demands in  $\text{Init}(w)$ . More specifically, it maintains the invariant that

$$|\text{Init}(w) \cap \text{Ball}(w, \frac{m(w)}{\psi})| \cdot m(w) \leq \beta f \quad (1)$$

After the algorithm has updated its configuration, it assigns the new demand to the nearest facility. We always distinguish between the arrival and the assignment time of a demand because the algorithm's configuration may have changed in between.

If the demands considered by IFL occupy  $m$  different locations, IFL can be implemented in  $O(nm|F_{\max}|)$  time and  $O(\min\{n, m|F_{\max}|\})$  space, where  $|F_{\max}|$  is the maximum number of facilities in  $F$  at any point in time. The remaining of this section is devoted to the proof of the following theorem.

**Theorem 1.** *For every  $x \geq 18$ ,  $\beta \geq \frac{4(x+1)}{x-8}$ , and  $\psi \in [4, 5]$ , IFL achieves a constant performance ratio for Facility Location with uniform facility costs.*

**Preliminaries.** For an arbitrary fixed sequence of demands, we compare the algorithm's cost with the cost of the optimal facility configuration, denoted by  $F^*$ . We reserve the term (optimal) *center* for the optimal facilities in  $F^*$  and the term *facility* for the algorithm's facilities in  $F$ . In the optimal solution, each demand  $u$  is assigned to the nearest center in  $F^*$ , denoted by  $c_u$ . We use the clustering induced by  $F^*$  to map the demands and the algorithm's facilities to the optimal centers. In particular, a demand  $u$  is always mapped to  $c_u$ . Similarly, a facility  $w$  is mapped to the nearest optimal center, denoted by  $c_w$ . Also, let  $d_u^* = d(c_u, u)$  be the optimal assignment cost of  $u$ , let  $\text{Fac}^* = |F^*|f$  be the optimal facility cost, and let  $\text{Asg}^* = \sum_u d_u^*$  be the optimal assignment cost.



To refer to the algorithm's configuration at the moment a demand is considered (resp. facility opens), we use the demand's (resp. facility's) identifier as a subscript. We distinguish between the algorithm's configuration at the demand's arrival and assignment time using plain symbols to refer to the former and primed symbols to refer to the latter time. For example, for a demand  $u$ ,  $F_u$  (resp.  $F'_u$ ) is the facility configuration at  $u$ 's arrival (resp. assignment) time. Similarly, for a facility  $w$ ,  $F_w$  (resp.  $F'_w$ ) is the facility configuration just before (resp. after)  $w$  opens.

For each facility  $w$ , we let  $B_w \equiv \text{Ball}(w, d(F_w, w)/x) \cap L_w$  denote the unsatisfied neighborhood contributing its potential towards opening  $w$  (i.e., if  $u$  is the demand opening  $w$ , then  $B_w = B_u$ ). When an existing facility  $w$  is merged with a new facility  $w'$ , the existing facility  $w$  is closed and the demands currently assigned to  $w$  are re-assigned to the new facility  $w'$  (and not the other way around).

**Basic Properties.** The following lemmas establish the basic properties of IFL.

**Lemma 1.** *For every facility  $w$  mapped to  $c_w$ ,  $d(c_w, w) \leq d(F_w, c_w)/3$ .*

*Proof Sketch.* Each new facility is opened by a small unsatisfied neighborhood with a large potential. If there were no optimal centers close to such a neighborhood, the cost of the optimal solution could decrease by opening a new center.

For a formal proof, we assume that there is a facility  $w$  such that  $d(c_w, w) > d(F_w, c_w)/3$ . Since  $B_w \subseteq \text{Ball}(w, d(F_w, w)/x)$ , we can show that for each  $u \in B_w$ ,  $d(u, w) < \frac{4}{x-4}d_u^*$  and  $d(F_w, u) < \frac{4(x+1)}{x-4}d_u^*$ . Using  $\text{Pot}(B_w) \geq \beta f$ , we conclude that for  $\beta$  appropriately large,  $f + \sum_{u \in B_w} d(u, w) < \sum_{u \in B_w} d_u^*$ , which contradicts to the optimality of  $F^*$ .  $\square$

**Lemma 2.** *For every facility  $w$ , there will always exist a facility in  $\text{Ball}(w, \frac{x}{x-3}m(w))$  and every demand currently assigned to  $w$  will remain assigned to a facility in  $\text{Ball}(w, \frac{x}{x-3}m(w))$ .*

*Proof Sketch.* The lemma holds because every time a facility  $w$  is merged with a new facility  $w'$ , the merge radius of  $w'$  is a fraction of the merge radius of  $w$ . Formally, if  $w$  is merged with a new facility  $w'$ , it must be  $d(w, w') \leq m(w)$  and  $m(w') \leq \frac{3}{x}d(w, w')$ . Hence,  $\text{Ball}(w', \frac{x}{x-3}m(w')) \subseteq \text{Ball}(w, \frac{x}{x-3}m(w))$ .  $\square$

**Facility Cost.** We distinguish between *supported* and *unsupported facilities*: A facility  $w$  is supported if  $\text{Asg}^*(B_w) = \sum_{u \in B_w} d_u^* \geq \frac{\beta}{3x}f$ , and unsupported otherwise. The cost of each supported facility  $w$  can be directly charged to the optimal assignment cost of the demands in  $B_w$ . Since each demand contributes to the facility cost at most once, the total cost of supported facilities is at most  $\frac{3x}{\beta} \text{Asg}^*$ . On the other hand, the merge ball of each unsupported facility  $w$  mapped to the optimal center  $c_w$  includes a sufficiently large area around  $c_w$ . Thus, each unsupported facility  $w$  mapped to  $c_w$  is merged with the first new facility  $w'$  also mapped to  $c_w$  because  $w'$  is included in  $w$ 's merge ball. Consequently, there can be at most one unsupported facility mapped to each optimal center. Hence, the total cost of unsupported facilities never exceeds  $\text{Fac}^*$  and the algorithm's facility cost is at most  $\text{Fac}^* + \frac{3x}{\beta} \text{Asg}^*$ .

**Lemma 3.** *Let  $w$  be an unsupported facility mapped to an optimal center  $c_w$ , and let  $w'$  be a new facility also mapped to  $c_w$ . If  $w'$  opens while  $w$  is still open, then  $w$  is merged with  $w'$ .*

*Proof Sketch.* By Lemma 1, it must be  $d(c_w, w') \leq d(F_{w'}, c_w)/3 \leq d(c_w, w)/3$ , because  $w'$  is mapped to  $c_w$  and  $w \in F_{w'}$  by hypothesis. We can also prove that  $m(w) \geq 3d(c_w, w)/2$ . This implies the lemma because  $d(w, w') \leq d(c_w, w) + d(c_w, w') \leq 4d(c_w, w)/3 \leq m(w)$  and  $w$  must be merged with  $w'$ .  $\square$

**Assignment Cost.** We start with a sketch of the analysis in the special case that  $F^*$  consists of a single optimal center, denoted by  $c$ . The same ideas can be applied to the case that some optimal centers are very close to each other and isolated from the remaining centers in  $F^*$  (cf. *isolated active coalitions*).

Let  $w$  be the facility which is currently the nearest facility to  $c$ . By Lemma 1,  $w$  stops being the nearest facility to  $c$  as soon as a new facility  $w'$  opens. By Lemma 3, if  $w$  is an unsupported facility, it is merged with  $w'$ . The main idea is that as long as the algorithm keeps merging existing facilities with new ones (which are significantly closer to  $c$ ), the algorithm's facility configuration converges rapidly to  $c$  and the algorithm's assignment cost converges to the optimal assignment cost. In addition, the rule for opening new facilities ensures that the algorithm's assignment cost for the demands arriving after  $w$ 's and before  $w'$ 's opening remains within a constant factor from the optimal cost. A simple induction yields that the assignment cost for the demands arriving as long as existing facilities are merged with new ones (cf. *good inner demands*) remains within a constant factor from the optimal cost.

However, if  $w$  is a supported facility, it may not be merged with the new facility  $w'$ . Then, the algorithm is charged with an irrevocable cost because the chain of merge operations converging to the optimal center has come to an end. This cost accounts for the assignment cost of the demands whose facility has stopped converging to  $c$  (cf. *bad inner demands*). This cost is charged to the optimal assignment cost of the demands in  $B_w$ , which is at least  $\frac{\beta}{3x}f$  since  $w$  is a supported facility.

For the analysis of non-isolated sets of optimal centers (cf. *non-isolated active coalitions*), it is crucial that Lemma 2 implies a notion of distance (cf. *configuration distance*) according to which the algorithm's configuration converges monotonically to each point of the metric space. The analysis is based on the fact that a set of optimal centers can be non-isolated only if its distance to the algorithm's configuration lies in a relatively short interval. This implies that the case dominating the analysis and determining the algorithm's performance ratio is that of isolated sets of optimal centers.

We proceed to formally define the basic notions used in the analysis of the algorithm's assignment cost. In the following,  $\lambda = 3x + 2$ ,  $\rho = (\psi + 2)(\lambda + 2)$ , and  $\gamma = 12\rho$  are appropriately chosen constants.

**Configuration Distance.** For an optimal center  $c \in F^*$  and a facility  $w \in F$ , the *configuration distance* between  $c$  and  $w$ , denoted by  $g(c, w)$ , is  $g(c, w) = d(c, w) + \frac{x}{x-3}m(w)$ . For an optimal center  $c \in F^*$ , the *configuration distance* of

$c$ , denoted by  $g(c)$ , is  $g(c) = \min_{w \in F} \{g(c, w)\} = \min_{w \in F} \{d(c, w) + \frac{x}{x-3} m(w)\}$ . The configuration distance  $g(c)$  is non-increasing with time and there always exists a facility within a distance of  $g(c)$  from  $c$  (Lemma 2).

*Coalitions.* A set of optimal centers  $K \subseteq F^*$  with representative  $c_K \in K$  forms a *coalition* as long as  $g(c_K) > \rho D(K)$ . A coalition  $K$  becomes *broken* as soon as  $g(c_K) \leq \rho D(K)$ . A coalition  $K$  is *isolated* if  $g(c_K) \leq \text{sep}(K)/3$  and *non-isolated* otherwise, where  $\text{sep}(K) = d(K, F^* \setminus K)$  denotes the distance separating the optimal centers in  $K$  from the remaining optimal centers. Intuitively, as long as  $K$ 's diameter is much smaller than  $g(c_K)$  ( $K$  is a coalition), the algorithm behaves as if  $K$  was a single optimal center. If the algorithm is bound to have a facility which is significantly closer to  $K$  than any optimal center not in  $K$  ( $K$  is isolated), then as far as  $K$  is concerned, the algorithm behaves as if there were no optimal centers outside  $K$ .

A *hierarchical decomposition*  $\mathcal{K}$  of  $F^*$  is a complete laminar set system<sup>1</sup> on  $F^*$ . Every hierarchical decomposition of  $F^*$  contains at most  $2|F^*| - 1$  sets. Given a hierarchical decomposition  $\mathcal{K}$  of  $F^*$ , we can fix an arbitrary representative  $c_K$  for each  $K \in \mathcal{K}$  and regard  $\mathcal{K}$  as a system of coalitions which hierarchically cover  $F^*$ . Formally, given a hierarchical decomposition  $\mathcal{K}$  of  $F^*$  and the current algorithm's configuration, a set  $K \in \mathcal{K}$  is an *active coalition* if  $K$  is still a coalition (i.e.,  $g(c_K) > \rho D(K)$ ), while every superset of  $K$  in  $\mathcal{K}$  has become broken (i.e., for every  $K' \in \mathcal{K}, K \subset K', g(c_{K'}) \leq \rho D(K')$ ). The current algorithm's configuration induces a collection of active coalitions which form a partitioning of  $F^*$ . Since  $g(c_K)$  is non-increasing, no coalition which has become broken (resp. isolated) can become active (resp. non-isolated) again.

Let  $D_N(K) \equiv \max\{D(K), \frac{1}{3\rho} \text{sep}(K)\}$ . By definition,  $K$  becomes either isolated or broken as soon as  $g(c_K) \leq \rho D_N(K)$ . Using [6, Lemma 1], we can show that there is a hierarchical decomposition of  $F^*$  such that no coalition  $K$  becomes active before  $g(c_K) < (\rho+1)\gamma^2 D_N(K)$ . In the following, we assume that the set of active coalitions is given by a fixed hierarchical decomposition  $\mathcal{K}$  of  $F^*$  such that for every non-isolated active coalition  $K$ ,  $\rho D_N(K) < g(c_K) < (\rho+1)\gamma^2 D_N(K)$ .

Each new demand  $u$  is mapped to the unique active coalition containing  $c_u$  when  $u$  arrives. Each new facility  $w$  is mapped to the unique active coalition containing  $c_w$  just before  $w$  opens. For an isolated active coalition  $K$ , we let  $w_K$  denote the *nearest facility* to  $K$ 's representative  $c_K$  at any given point in time. In other words,  $w_K$  is a function always associating the isolated active coalition  $K$  with the facility in  $F$  which is currently the nearest facility to  $c_K$ .

*Final Assignment Cost.* Let  $u$  be a demand currently assigned to a facility  $w$  with merge radius  $m(w)$ . The *final assignment cost* of  $u$ , denoted by  $\bar{d}_u$ , is equal to  $\min\{d(u, w) + \frac{x}{x-3} m(w), (1 + \frac{1}{\lambda}) \max\{d(c_K, w), \lambda D(K)\} + d_u^*\}$  if  $u$  is mapped to an isolated active coalition  $K$  and  $w$  is currently the nearest facility to  $c_K$ , and equal to  $d(u, w) + \frac{x}{x-3} m(w)$  otherwise. If  $u$  is currently assigned to a facility  $w$ , it will

<sup>1</sup> A set system is *laminar* if it contains no intersecting pair of sets. The sets  $K, K'$  form an *intersecting pair* if neither of  $K \setminus K', K' \setminus K$  and  $K \cap K'$  are empty. A laminar set system on  $F^*$  is *complete* if it contains  $F^*$  and every singleton set  $\{c\}$ ,  $c \in F^*$ .

remain assigned to a facility in  $\text{Ball}(w, \frac{x}{x-3} m(w))$  (Lemma 2). We can also prove that if  $u$  is mapped to an isolated active coalition  $K$  and is currently assigned to  $w_K$ , then  $u$ 's assignment cost will never exceed  $(1 + \frac{1}{\lambda}) \max\{d(c_K, w_K), \lambda D(K)\} + d_u^*$ . Therefore, the final assignment cost of  $u$  according to the current algorithm's configuration is an upper bound on its actual assignment cost in the future.

*Inner and Outer Demands.* A demand  $u$  mapped to a non-isolated active coalition  $K$  is *inner* if  $d_u^* < D_N(K)$ , and *outer* otherwise. A demand  $u$  mapped to an isolated active coalition  $K$  is *inner* if  $d_u^* < \frac{1}{\lambda} \max\{d(c_K, w'_K), \lambda D(K)\}$ , and *outer* otherwise. In this definition,  $w'_K$  denotes the nearest facility to  $c_K$  at  $u$ 's assignment time.

We can prove that the final assignment cost of each outer demand is within a constant factor from its optimal assignment cost. From now on, we entirely focus on bounding the total assignment cost of inner demands throughout the execution of the algorithm.

*Good and Bad Inner Demands.* At any point in time, the set of *good demands* of an *isolated* active coalition  $K$ , denoted by  $G_K$ , consists of the inner demands of  $K$  which have *always* (i.e., from their assignment time until the present time) been assigned to  $w_K$  (i.e., the nearest facility to  $c_K$ ).  $G_K$  is empty as long as  $K$  is either not active or non-isolated. We call *bad* every inner demand of  $K$  which is not good.

Each new inner demand mapped to an isolated active coalition  $K$  is initially assigned to the nearest facility to  $c_K$  because this facility is much closer to  $c_K$  than any other facility (see also Lemma 1) and inner demands are included in a small ball around  $c_K$ . Hence, each new inner demand mapped to  $K$  becomes good. An inner demand remains good as long as its assignment cost converges to its optimal assignment cost. In particular, a good inner demand becomes bad as soon as either  $K$  becomes broken or the location of the nearest facility to  $c_K$  changes and the facility at the former location  $w_K$  is not merged with the facility at the new location  $w'_K$ . A bad demand can never become good again.

With the exception of good demands, each demand is *irrevocably* charged with its final assignment cost at its assignment time. We keep track of the actual assignment cost of good demands until they become bad. This is possible because the good demands of an isolated active coalition  $K$  are always assigned to the nearest facility to  $c_K$ . Good demands are irrevocably charged with their final assignment cost at the moment they become bad.

*Isolated Coalitions.* We use the following lemma to bound the assignment cost of the inner demands mapped to an isolated active coalition  $K$ .

**Lemma 4.** *Let  $K$  be an isolated active coalition. The total actual and the total final assignment cost of the good inner demands of  $K$  is:*

$$\sum_{u \in G_K} d(u, w_K) < 2\beta f + 3 \sum_{u \in G_K} d_u^* \quad \text{and} \quad \sum_{u \in G_K} \bar{d}_u < 4.5\beta f + 7 \sum_{u \in G_K} d_u^*$$

*Proof Sketch.* We sketch the proof of the first inequality. The second inequality is derived from the first one using the definition of the final assignment cost.

The proof is by induction over a sequence of merge operations where the former nearest facility to  $c_K$  is merged with the new nearest facility to  $c_K$ . Let

$w$  be the nearest facility to  $c_K$ , i.e.,  $w_K = w$ . We can show that for any isolated coalition  $K$ , the location of the nearest facility to  $c_K$  cannot change until a new facility mapped to  $K$  opens. Let  $w'$  be the next facility mapped to  $K$  and let  $G_K$  be the set of good demands just before  $w'$  opens. By Lemma 1, the location of the nearest facility to  $c_K$  changes from  $w_K = w$  to  $w'_K = w'$  as soon as  $w'$  opens. We inductively assume that the inequality holds just before  $w'$  opens, and show that it remains valid until either the location of the nearest facility to  $c_K$  changes again or  $K$  becomes broken.

If  $w$  is not merged with  $w'$ , the set of good demands becomes empty and the inequality holds trivially. If  $w$  is merged with  $w'$ , we can apply Lemma 1 and show that for every  $u \in G_K$ ,  $d(u, w') \leq \frac{1}{2}d(u, w) + \frac{3}{2}d_u^*$ . Just after  $w$  has been merged with  $w'$ , the set of good demands remains  $G_K$ , but each good demand is now assigned to  $w'$ . Hence, we can apply the inductive hypothesis and the inequality above and prove that  $\sum_{u \in G_K} d(u, w') \leq \beta f + 3 \sum_{u \in G_K} d_u^*$ .

As long as  $w'$  remains the nearest facility to  $c_K$  and  $K$  remains an isolated active coalition, each new inner demand mapped to  $K$  is initially assigned to  $w'$  and becomes a good demand. The rule for opening new facilities and Ineq. (1) imply that the initial assignment cost of these demands is at most  $\beta f$ .  $\square$

As long as  $K$  remains an isolated active coalition, it holds a credit of  $O(\beta f)$  which absorbs the additive term of  $2\beta f$  corresponding to the part of the actual assignment cost of good inner demands which cannot be charged to their optimal assignment cost. The good inner demands of  $K$  are charged with their final assignment cost as soon as they become bad and  $G_K$  becomes empty. If  $G_K$  becomes empty because  $K$  becomes broken, the additive term of  $4.5\beta f$  in the final assignment cost of the demands becoming bad is charged to  $K$ 's credit. Otherwise,  $G_K$  becomes empty because the location of the nearest facility to  $c_K$  has changed and the facility  $w$  at the previous location  $w_K$  is not merged with the new facility  $w'$  at the new location  $w'_K$ . By Lemma 3, the facility  $w$  must be a supported facility. Hence, the additive term of  $4.5\beta f$  can be charged to the optimal assignment cost of the demands in  $B_w$ , since  $3x \text{Asg}^*(B_w) \geq \beta f$ . Lemma 1 implies that each supported facility is charged with the final assignment cost of some good demands which become bad at most once.

*Non-isolated Coalitions.* The inner demands mapped to non-isolated active coalitions are irrevocably charged with their final assignment cost at their assignment time. To bound their total final assignment cost, we develop a standard potential function argument based on the notion of *unsatisfied inner demands*. The set of unsatisfied inner demands of a non-isolated active coalition  $K$ , denoted by  $N_K$ , consists of the inner demands of  $K$  which are currently unsatisfied, i.e., in the set  $L$ .  $N_K$  is empty as long as  $K$  is either isolated or not active.

**Lemma 5.** *For every non-isolated active coalition  $K$ ,  $|N_K| \cdot g(c_K) \leq (\psi + 4)\gamma^2\beta f$ .*

As long as  $K$  remains a non-isolated active coalition,  $c_K$  has a deficit of  $5|N_K| \cdot g(c_K)$ , which, by Lemma 5, never exceeds  $5(\psi + 4)\gamma^2\beta f$ . If a new demand  $u$  is an inner demand of  $K$  and remains unsatisfied at its assignment time, we

can prove that its final assignment cost is at most  $5g'_u(c_K)$  (we recall that  $g'_u(c_K)$  denotes the configuration distance of  $c_K$  at  $u$ 's assignment time). Since  $u$  is added to  $N_K$ , the increase in the deficit of  $c_K$  compensates for the final assignment cost of  $u$ .

The deficit of  $c_K$  is 0 before  $K$  becomes active and after  $K$  has become either isolated or broken. Hence, the total final assignment cost of the inner demands mapped to  $K$  while  $K$  remains a non-isolated active coalition does not exceed the total decrease in the deficit of  $c_K$ . Each time the configuration distance  $g(c_K)$  decreases but the set of unsatisfied inner demands  $N_K$  remains unaffected, the deficit of  $c_K$  decreases by a factor of  $\ln(\frac{g(c_K)}{g'(c_K)})$ . We can also prove that each time opening a new facility causes some demands to become satisfied and be removed from  $N_K$ ,  $g(c_K)$  decreases by a factor of 3. Thus, the deficit of  $c_K$  again decreases by a factor of  $\ln(\frac{g(c_K)}{g'(c_K)})$ . A non-isolated active coalition  $K$  becomes either isolated or broken no later than  $g(c_K)$  has decreased by a factor of  $(1 + \frac{1}{\rho})\gamma^2$ . Therefore, the total decrease in the deficit of  $c_K$  is at most  $O(5(\psi + 4)\gamma^2\beta \ln((1 + \frac{1}{\rho})\gamma^2)f)$ . We increase this cost by  $O(\beta \ln((1 + \frac{1}{\rho})\gamma^2)f)$  to take into account the final assignment cost of the inner demands of  $K$  which open a new facility and do not remain unsatisfied at their assignment time.

**Concluding the Proof of Theorem 1.** Let  $u_1, \dots, u_n$  be the demand sequence considered by IFL. Putting everything together, we conclude that after the demand  $u_j$  has been considered, the facility cost of IFL does not exceed  $a_1 \text{Fac}^* + b_1 \text{Asg}_j^*$  and the assignment cost of IFL does not exceed  $a_2 \text{Fac}^* + b_2 \text{Asg}_j^*$ , where  $\text{Asg}_j^* = \sum_{i=1}^j d_{u_i}^*$ , and  $a_1 = 1$ ,  $a_2 = 2\beta \ln(3\gamma^2)(5(\psi + 4)\gamma^2 + 3)$ ,  $b_1 = 3x/\beta$ , and  $b_2 = 4((\rho + 1)\gamma^2 + 2) + 14x$ . With a more careful analysis, we can improve  $a_2$  to  $4\beta \log(\gamma)(12(\psi + 2) + 3)$  and  $b_2$  to  $(\lambda + 2)(8\psi + 25)$ .

### 3 An Incremental Algorithm for $k$ -Median

The incremental algorithm for  $k$ -Median is based on the following standard lemma. We recall that  $a_1, a_2, b_1$ , and  $b_2$  denote the constants in the performance ratio of IFL.

**Lemma 6.** *Let  $\text{Asg}^*$  be the optimal assignment cost for an instance of  $k$ -Median. For any  $\Lambda > 0$ , IFL with facility cost  $f = \frac{b_2}{a_2} \frac{\Lambda}{k}$  maintains a solution of assignment cost no greater than  $(a_2 + b_2)\text{Asg}^* + b_2\Lambda$  with no more than  $(a_1 + a_2 \frac{b_1}{b_2} \frac{\text{Asg}^*}{\Lambda})k$  medians.*

The deterministic version of the incremental algorithm operates in phases using IFL as a building block. Phase  $i$  is characterized by an upper bound  $\Lambda_i$  on the optimal assignment cost for the demands considered in the current phase. The algorithm invokes IFL with facility cost  $f_i = \frac{b_2}{a_2} \frac{\Lambda_i}{k}$ . By Lemma 6, as soon as either the number of medians exceeds  $\nu k$  or the assignment cost exceeds  $\mu \Lambda_i$ , where  $\nu, \mu$  are appropriately chosen constants, we can be sure that the optimal cost has also exceeded  $\Lambda_i$ . Then, the algorithm merges the medians produced by

the current phase with the medians produced by the previous phases, increases the upper bound by a constant factor, and proceeds with the next phase.

**Theorem 2.** *There exists a deterministic incremental algorithm for  $k$ -Median which achieves a constant performance ratio using  $O(k)$  medians.*

The randomized version also proceeds in phases. Phase  $i$  invokes the algorithm PARA\_CLUSTER [4] with facility cost  $\hat{f}_i = \frac{\Lambda_i}{k(\log n + 1)}$  to generate a modified instance which can be represented in a space efficient manner. The modified instance contains the same number of demands, which now occupy only  $O(k \log^2 n)$  different gathering points. Then, the algorithm invokes IFL with facility cost  $f_i = \frac{b_2}{a_2} \frac{\Lambda_i}{k}$  to cluster the modified instance.

For an incremental implementation, each new demand is first moved to a gathering point. Then, a new demand located at the corresponding gathering point is given to IFL, which assigns it to a median. Both actions are completed before the next demand is considered. The current phase ends if either the number of gathering points, the gathering cost, the number of medians, or the assignment cost on the modified instance become too large. We should emphasize that IFL treats the demands moved to the same gathering point as *different demands* and may put them in different clusters.

In contrast to the deterministic version which does not require any advance knowledge of  $n$ , the randomized version needs to know  $\log n$  in advance.

**Theorem 3.** *There exists a randomized incremental algorithm for  $k$ -Median which runs in  $O(nk^2 \log^2 n)$  time and  $O(k^2 \log^2 n)$  space and achieves a constant performance ratio whp. using  $O(k)$  medians.*

## References

1. A. Anagnostopoulos, R. Bent, E. Upfal, and P. Van Hentenryck. A Simple and Deterministic Competitive Algorithm for Online Facility Location. TR CS-03-16, Brown University, 2003.
2. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
3. M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental Clustering and Dynamic Information Retrieval. In *Proc. of STOC '97*, pp. 626–635, 1997.
4. M. Charikar, L. O'Callaghan, and R. Panigrahy. Better Streaming Algorithms for Clustering Problems. In *Proc. of STOC '03*, pp. 30–39, 2003.
5. M. Charikar and R. Panigrahy. Clustering to Minimize the Sum of Cluster Diameters. In *Proc. of STOC '01*, pp. 1–10, 2001.
6. D. Fotakis. On the Competitive Ratio for Online Facility Location. In *Proc. of ICALP '03*, LNCS 2719, pp. 637–652, 2003.
7. D. Fotakis. Incremental Algorithms for Facility Location and  $k$ -Median. Available from <http://users.auth.gr/~fotakis/data/incremen.pdf>, 2004.
8. S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering Data Streams. In *Proc. of FOCS '00*, pp. 359–366, 2000.
9. M.R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on Data Streams. TR 1998-011, DEC Systems Research Center, 1998.
10. A. Meyerson. Online Facility Location. In *Proc. of FOCS '01*, pp. 426–431, 2001.



# Dynamic Shannon Coding

Travis Gagie

Department of Computer Science  
University of Toronto  
travis@cs.toronto.edu

**Abstract.** We present a new algorithm for dynamic prefix-free coding, based on Shannon coding. We give a simple analysis and prove a better upper bound on the length of the encoding produced than the corresponding bound for dynamic Huffman coding. We show how our algorithm can be modified for efficient length-restricted coding, alphabetic coding and coding with unequal letter costs.

## 1 Introduction

Prefix-free coding is a well-studied problem in data compression and combinatorial optimization. For this problem, we are given a string  $S = s_1 \cdots s_m$  drawn from an alphabet of size  $n$  and must encode each character by a self-delimiting binary codeword. Our goal is to minimize the length of the entire encoding of  $S$ . For static prefix-free coding, we are given all of  $S$  before we start encoding and must encode every occurrence of the same character by the same codeword. The assignment of codewords to characters is recorded as a preface to the encoding. For dynamic prefix-free coding, we are given  $S$  character by character and must encode each character before receiving the next one. We can use a different codeword for different occurrences of the same character, we do not need a preface to the encoding and the assignment of codewords to characters cannot depend on the suffix of  $S$  not yet encoded.

The best-known algorithms for static coding are by Shannon [15] and Huffman [8]. Shannon's algorithm uses at most  $(H + 1)m + O(n \log n)$  bits to encode  $S$ , where

$$H = \sum_{a \in S} \frac{\#_a(S)}{m} \log \left( \frac{m}{\#_a(S)} \right)$$

is the empirical entropy of  $S$  and  $\#_a(S)$  is the number of occurrences of the character  $a$  in  $S$ . By  $\log$  we mean  $\log_2$ . Shannon proved a lower bound of  $Hm$  bits for all coding algorithms, whether or not they are prefix-free. Huffman's algorithm produces an encoding that, excluding the preface, has minimum length. The total length is  $(H + r)m + O(n \log n)$  bits, where  $0 \leq r < 1$  is a function of the character frequencies in  $S$  [3].

Both algorithms assign codewords to characters by constructing a *code-tree*, that is, a binary tree whose left and right edges are labelled by 0's and 1's, respectively, and whose leaves are labelled by the distinct characters in  $S$ . The



codeword assigned to a character  $a$  in  $S$  is the sequence of edge labels on the path from the root to the leaf labelled  $a$ . Shannon's algorithm builds a code-tree in which, for  $a \in S$ , the leaf labelled  $a$  is of depth at most  $\lceil \log(m/\#_a(S)) \rceil$ . Huffman's algorithm builds a Huffman tree for the frequencies of the characters in  $S$ . A *Huffman tree* for a sequence of weights  $w_1, \dots, w_n$  is a binary tree whose leaves, in some order, have weights  $w_1, \dots, w_n$  and that, among all such trees, minimizes the weighted external path length. To build a Huffman tree for  $w_1, \dots, w_n$ , we start with  $n$  trees, each consisting of just a root. At each step, we make the two roots with smallest weights,  $w_i$  and  $w_j$ , into the children of a new root with weight  $w_i + w_j$ .

A *minimax tree* for a sequence of weights  $w_1, \dots, w_n$  is a binary tree whose leaves, in some order, have weights  $w_1, \dots, w_n$  and that, among all such trees, minimizes the maximum sum of any leaf's weight and depth. Golumbic [7] gave an algorithm, similar to Huffman's, for constructing a minimax tree. The difference is that, when we make the two roots with smallest weights,  $w_i$  and  $w_j$ , into the children of a new root, that new root has weight  $\max(w_i, w_j) + 1$  instead of  $w_i + w_j$ . Notice that, if there exists a binary tree whose leaves, in some order, have depths  $d_1, \dots, d_n$ , then a minimax tree  $T$  for  $-d_1, \dots, -d_n$  is such a tree and, more generally, the depth of each node in  $T$  is bounded above by the negative of its weight. So we can construct a code-tree for Shannon's algorithm by running Golumbic's algorithm, starting with roots labelled by the distinct characters in  $S$ , with the root labelled  $a$  having weight  $-\lceil \log(m/\#_a(S)) \rceil$ .

Both Shannon's algorithm and Huffman's algorithm have three phases: a first pass over  $S$  to count the occurrences of each distinct character, an assignment of codewords to the distinct characters in  $S$  (recorded as a preface to the encoding) and a second pass over  $S$  to encode each character in  $S$  using the assigned codeword. The first phase takes  $O(m)$  time, the second  $O(n \log n)$  time and the third  $O((H + 1)m)$  time.

For any static algorithm  $\mathcal{A}$ , there is a simple dynamic algorithm that recomputes the code-tree from scratch after reading each character. Specifically, for  $i = 1 \dots m$ :

1. We keep a running count of the number of occurrences of each distinct character in the current prefix  $s_1 \dots s_{i-1}$  of  $S$ .
2. We compute the assignment of codewords to characters that would result from applying  $\mathcal{A}$  to  $\perp s_1 \dots s_{i-1}$ , where  $\perp$  is a special character not in the alphabet.
3. If  $s_i$  occurs in  $s_1 \dots s_{i-1}$ , then we encode  $s_i$  as the codeword  $c_i$  assigned to that character.
4. If  $s_i$  does not occur in  $s_1 \dots s_{i-1}$ , then we encode  $s_i$  as the concatenation  $c_i$  of the codeword assigned to  $\perp$  and the binary representation of  $s_i$ 's index in the alphabet.

We can later decode character by character. That is, we can recover  $s_1 \dots s_i$  as soon as we have received  $c_1 \dots c_i$ . To see why, assume that we have recovered  $s_1 \dots s_{i-1}$ . Then we can compute the assignment of codewords to characters that  $\mathcal{A}$  used to encode  $s_i$ . Since  $\mathcal{A}$  is prefix-free,  $c_i$  is the only codeword in this

assignment that is a prefix of  $c_i \cdots c_m$ . Thus, we can recover  $s_i$  as soon as  $c_i$  has been received. This takes the same amount of time as encoding  $s_i$ .

Faller [4] and Gallager [6] independently gave a dynamic coding algorithm based on Huffman's algorithm. Their algorithm is similar to, but much faster than, the simple dynamic algorithm obtained by adapting Huffman's algorithm as described above. After encoding each character of  $S$ , their algorithm merely updates the Huffman tree rather than rebuilding it from scratch. Knuth [10] implemented their algorithm so that it uses time proportional to the length of the encoding produced. For this reason, it is sometimes known as Faller-Gallager-Knuth coding; however, it is most often called *dynamic Huffman coding*. Milidiú, Laber, and Pessoa [14] showed that this version of dynamic Huffman coding uses fewer than  $2m$  more bits to encode  $S$  than Huffman's algorithm. Vitter [17] gave an improved version that he showed uses fewer than  $m$  more bits than Huffman's algorithm. These results imply Knuth's and Vitter's versions use at most  $(H + 2 + r)m + O(n \log n)$  and  $(H + 1 + r)m + O(n \log n)$  bits to encode  $S$ , but it is not clear whether these bounds are tight. Both algorithms use  $O((H + 1)m)$  time.

In this paper, we present a new dynamic algorithm, *dynamic Shannon coding*. In Section 2, we show that the simple dynamic algorithm obtained by adapting Shannon's algorithm as described above, uses at most  $(H + 1)m + O(n \log m)$  bits and  $O(mn \log n)$  time to encode  $S$ . Section 3 contains our main result, an improved version of dynamic Shannon coding that uses at most  $(H + 1)m + O(n \log m)$  bits to encode  $S$  and only  $O((H + 1)m + n \log^2 m)$  time. The relationship between Shannon's algorithm and this algorithm is similar to that between Huffman's algorithm and dynamic Huffman coding, but our algorithm is much simpler to analyze than dynamic Huffman coding.

In Section 4, we show that dynamic Shannon coding can be applied to three related problems. We give algorithms for dynamic length-restricted coding, dynamic alphabetic coding and dynamic coding with unequal letter costs. Our algorithms have better bounds on the length of the encoding produced than were previously known. For length-restricted coding, no codeword can exceed a given length. For alphabetic coding, the lexicographic order of the codewords must be the same as that of the characters.

Throughout, we make the common simplifying assumption that  $m \geq n$ . Our model of computation is the unit-cost word RAM with  $\Omega(\log m)$ -bit words. In this model, ignoring space required for the input and output, all the algorithms mentioned in this paper use  $O(|\{a : a \in S\}|)$  words, that is, space proportional to the number of distinct characters in  $S$ .

## 2 Analysis of Simple Dynamic Shannon Coding

In this section, we analyze the simple dynamic algorithm obtained by repeating Shannon's algorithm after each character of the string  $\perp s_1 \cdots s_m$ , as described in the introduction. Since the second phase of Shannon's algorithm, assigning codewords to characters, takes  $O(n \log n)$  time, this simple algorithm

uses  $O(mn \log n)$  time to encode  $S$ . The rest of this section shows this algorithm uses at most  $(H+1)m + O(n \log m)$  bits to encode  $S$ .

For  $1 \leq i \leq m$  and each distinct character  $a$  that occurs in  $\perp s_1 \cdots s_{i-1}$ , Shannon's algorithm on  $\perp s_1 \cdots s_{i-1}$  assigns to  $a$  a codeword of length at most  $\lceil \log(i/\#_a(\perp s_1 \cdots s_{i-1})) \rceil$ . This fact is key to our analysis.

Let  $R$  be the set of indices  $i$  such that  $s_i$  is a repetition of a character in  $s_1 \cdots s_{i-1}$ . That is,  $R = \{i : 1 \leq i \leq m, s_i \in \{s_1, \dots, s_{i-1}\}\}$ . Our analysis depends on the following technical lemma.

**Lemma 1.**

$$\sum_{i \in R} \log \left( \frac{i}{\#_{s_i}(s_1 \cdots s_{i-1})} \right) \leq Hm + O(n \log m) .$$

*Proof.* Let

$$L = \sum_{i \in R} \log \left( \frac{i}{\#_{s_i}(s_1 \cdots s_{i-1})} \right) .$$

Notice that  $\sum_{i \in R} \log i < \sum_{i=1}^m \log i = \log(m!)$ . Also, for  $i \in R$ , if  $s_i$  is the  $j$ th occurrence of  $a$  in  $S$ , for some  $j \geq 2$ , then  $\log \#_{s_i}(s_1 \cdots s_{i-1}) = \log(j-1)$ . Thus,

$$\begin{aligned} L &= \sum_{i \in R} \log i - \sum_{i \in R} \log \#_{s_i}(s_1 \cdots s_{i-1}) \\ &< \log(m!) - \sum_{a \in S} \sum_{j=2}^{\#_a(S)} \log(j-1) \\ &= \log(m!) - \sum_{a \in S} \log(\#_a(S)!) + \sum_{a \in S} \log \#_a(S) . \end{aligned}$$

There are at most  $n$  distinct characters in  $S$  and each occurs at most  $m$  times, so  $\sum_{a \in S} \log \#_a(S) \in O(n \log m)$ . By Stirling's Formula,

$$x \log x - x \ln 2 < \log(x!) \leq x \log x - x \ln 2 + O(\log x) .$$

Thus,

$$L < m \log m - m \ln 2 - \sum_{a \in S} \left( \#_a(S) \log \#_a(S) - \#_a(S) \ln 2 \right) + O(n \log m) .$$

Since  $\sum_{a \in S} \#_a(S) = m$ ,

$$L < \sum_{a \in S} \#_a(S) \log \left( \frac{m}{\#_a(S)} \right) + O(n \log m) .$$

By definition, this is  $Hm + O(n \log m)$ . □

Using Lemma 1, it is easy to bound the number of bits that simple dynamic Shannon coding uses to encode  $S$ .

**Theorem 1.** *Simple dynamic Shannon coding uses at most  $(H + 1)m + O(n \log m)$  bits to encode  $S$ .*

*Proof.* If  $s_i$  is the first occurrence of that character in  $S$  (i.e.,  $i \in \{1, \dots, m\} - R$ ), then the algorithm encodes  $s_i$  as the codeword for  $\perp$ , which is at most  $\lceil \log m \rceil$  bits, followed by the binary representation of  $s_i$ 's index in the alphabet, which is  $\lceil \log n \rceil$  bits. Since there are at most  $n$  such characters, the algorithm encodes them all using  $O(n \log m)$  bits.

Now, consider the remaining characters in  $S$ , that is, those characters whose indices are in  $R$ . In total, the algorithm encodes these using at most

$$\sum_{i \in R} \left\lceil \log \left( \frac{i}{\#_{s_i}(\perp s_1 \cdots s_{i-1})} \right) \right\rceil < m + \sum_{i \in R} \log \left( \frac{i}{\#_{s_i}(s_1 \cdots s_{i-1})} \right)$$

bits. By Lemma 1, this is at most  $(H + 1)m + O(n \log m)$ .

Therefore, in total, this algorithm uses at most  $(H + 1)m + O(n \log m)$  bits to encode  $S$ .  $\square$

### 3 Dynamic Shannon Coding

This section explains how to improve simple dynamic Shannon coding so that it uses at most  $(H + 1)m + O(n \log m)$  bits and  $O((H + 1)m + n \log^2 m)$  time to encode the string  $S = s_1 \cdots s_m$ . The main ideas for this algorithm are using a dynamic minimax tree to store the code-tree, introducing “slack” in the weights and using background processing to keep the weights updated.

Gagie [5] showed that Faller's, Gallager's and Knuth's techniques for making Huffman trees dynamic can be used to make minimax trees dynamic. A *dynamic minimax tree*  $T$  supports the following operations:

- given a pointer to a node  $v$ , return  $v$ 's parent, left child, and right child (if they exist);
- given a pointer to a leaf  $v$ , return  $v$ 's weight;
- given a pointer to a leaf  $v$ , increment  $v$ 's weight;
- given a pointer to a leaf  $v$ , decrement  $v$ 's weight;
- and, given a pointer to a leaf  $v$ , insert a new leaf with the same weight as  $v$ .

In Gagie's implementation, if the depth of each node is bounded above by the negative of its weight, then each operation on a leaf with weight  $-d_i$  takes  $O(d_i)$  time. Next, we will show how to use this data structure for fast dynamic Shannon coding.

We maintain the invariant that, after we encode  $s_1 \cdots s_{i-1}$ ,  $T$  has one leaf labelled  $a$  for each distinct character  $a$  in  $\perp s_1 \cdots s_{i-1}$  and this leaf has weight between  $-\lceil \log((i+n)/\#_a(\perp s_1 \cdots s_{i-1})) \rceil$  and  $-\lceil \log(\max(i, n)/\#_a(\perp s_1 \cdots s_{i-1})) \rceil$ . Notice that applying Shannon's algorithm to  $\perp s_1 \cdots s_{i-1}$  results in a code-tree in which, for  $a \in \perp s_1 \cdots s_{i-1}$ , the leaf labelled  $a$  is of depth at most  $\lceil \log(i/\#_a(\perp s_1 \cdots s_{i-1})) \rceil$ . It follows that the depth of each node in  $T$  is bounded above by the negative of its weight.

Notice that, instead of having just  $i$  in the numerator, as we would for simple dynamic Shannon coding, we have at most  $i+n$ . Thus, this algorithm may assign slightly longer codewords to some characters. We allow this “slack” so that, after we encode each character, we only need to update the weights of at most two leaves. In the analysis, we will show that the extra  $n$  only affects low-order terms in the bound on the length of the encoding.

After we encode  $s_i$ , we ensure that  $T$  contains one leaf labelled  $s_i$  and this leaf has weight  $-\lceil \log((i+1+n)/\#_{s_i}(\perp s_1 \cdots s_i)) \rceil$ . First, if  $s_i$  is the first occurrence of that distinct character in  $S$  (i.e.,  $i \in \{1, \dots, m\} - R$ ), then we insert a new leaf labelled  $s_i$  into  $T$  with the same weight as the leaf labelled  $\perp$ . Next, we update the weight of the leaf labelled  $s_i$ . We consider this processing to be in the foreground.

In the background, we use a queue to cycle through the distinct characters that have occurred in the current prefix. For each character that we encode in the foreground, we process one character in the background. When we dequeue a character  $a$ , if we have encoded precisely  $s_1 \cdots s_i$ , then we update the weight of the leaf labelled  $a$  to be  $-\lceil \log((i+1+n)/\#_a(\perp s_1 \cdots s_i)) \rceil$ , unless it has this weight already. Since there are always at most  $n+1$  distinct characters in the current prefix ( $\perp$  and the  $n$  characters in the alphabet), this maintains the following invariant: For  $1 \leq i \leq m$  and  $a \in \perp s_1 \cdots s_{i-1}$ , immediately after we encode  $s_1 \cdots s_{i-1}$ , the leaf labelled  $a$  has weight between  $-\lceil \log((i+n)/\#_a(\perp s_1 \cdots s_{i-1})) \rceil$  and  $-\lceil \log(\max(i, n)/\#_a(\perp s_1 \cdots s_{i-1})) \rceil$ . Notice that  $\max(i, n) < i+n \leq 2\max(i, n)$  and  $\#_a(s_1 \cdots s_{i-1}) \leq \#_a(\perp s_1 \cdots s_i) \leq 2\#_a(s_1 \cdots s_{i-1}) + 1$ . Also, if  $s_i$  is the first occurrence of that distinct character in  $S$ , then  $\#_{s_i}(\perp s_1 \cdots s_i) = \#_{\perp}(\perp s_1 \cdots s_{i-1})$ . It follows that, whenever we update a weight, we use at most one increment or decrement.

Our analysis of this algorithm is similar to that in Section 2, with two differences. First, we show that weakening the bound on codeword lengths does not significantly affect the bound on the length of the encoding. Second, we show that our algorithm only takes  $O((H+1)m+n \log^2 m)$  time. Our analysis depends on the following technical lemma.

**Lemma 2.** *Suppose  $I \subseteq \mathbb{Z}^+$  and  $|I| \geq n$ . Then*

$$\sum_{i \in I} \log \left( \frac{i+n}{x_i} \right) \leq \sum_{i \in I} \log \left( \frac{i}{x_i} \right) + n \log(\max I + n) .$$

*Proof.* Let

$$L = \sum_{i \in I} \log \left( \frac{i+n}{x_i} \right) = \sum_{i \in I} \log \left( \frac{i}{x_i} \right) + \sum_{i \in I} \log \left( \frac{i+n}{i} \right) .$$

Let  $i_1, \dots, i_{|I|}$  be the elements of  $I$ , with  $0 < i_1 < \dots < i_{|I|}$ . Then  $i_j + n \leq i_{j+n}$ , so

$$\sum_{i \in I} \log \left( \frac{i+n}{i} \right) = \log \left( \frac{\left( \prod_{j=1}^{|I|-n} (i_j + n) \right) \left( \prod_{j=|I|-n+1}^{|I|} (i_j + n) \right)}{\left( \prod_{j=1}^n i_j \right) \left( \prod_{j=n+1}^{|I|} i_j \right)} \right)$$

$$\begin{aligned} &\leq \log \left( \frac{\left( \prod_{j=1}^{|I|-n} i_{j+n} \right) (\max I + n)^n}{1 \cdot \prod_{j=1}^{|I|-n} i_{j+n}} \right) \\ &= n \log(\max I + n) . \end{aligned}$$

Therefore,

$$L \leq \sum_{i \in I} \log \left( \frac{i}{x_i} \right) + n \log(\max I + n) . \quad \square$$

Using Lemmas 1 and 2, it is easy to bound the number of bits and the time dynamic Shannon coding uses to encode  $S$ , as follows.

**Theorem 2.** *Dynamic Shannon coding uses at most  $(H+1)m + O(n \log m)$  bits and  $O((H+1)m + n \log^2 m)$  time.*

*Proof.* First, we consider the length of the encoding produced. Notice that the algorithm encodes  $S$  using at most

$$\begin{aligned} &\sum_{i \in R} \left\lceil \log \left( \frac{i+n}{\#_{s_i}(\perp s_1 \cdots s_{i-1})} \right) \right\rceil + O(n \log m) \\ &\leq m + \sum_{i \in R} \log \left( \frac{i+n}{\#_{s_i}(s_1 \cdots s_{i-1})} \right) + O(n \log m) \end{aligned}$$

bits. By Lemmas 1 and 2, this is at most  $(H+1)m + O(n \log m)$ .

Now, we consider how long this algorithm takes. We will prove separate bounds on the processing done in the foreground and in the background.

If  $s_i$  is the first occurrence of that character in  $S$  (i.e.,  $i \in \{1, \dots, m\} - R$ ), then we perform three operations in the foreground when we encode  $s_i$ : we output the codeword for  $\perp$ , which is at most  $\lceil \log(i+n) \rceil$  bits; we output the index of  $s_i$  in the alphabet, which is  $\lceil \log n \rceil$  bits; and we insert a new leaf labelled  $s_i$  and update its weight to be  $-\lceil \log(i+1+n) \rceil$ . In total, these take  $O(\log(i+n)) \subseteq O(\log m)$  time. Since there are at most  $n$  such characters, the algorithm encodes them all using  $O(n \log m)$  time.

For  $i \in R$ , we perform at most two operations in the foreground when we encode  $s_i$ : we output the codeword for  $s_i$ , which is of length at most  $\lceil \log((i+n)/\#_{s_i}(s_1 \cdots s_{i-1})) \rceil$ ; and, if necessary, we increment the weight of the leaf labelled  $s_i$ . In total, these take  $O(\log((i+n)/\#_{s_i}(s_1 \cdots s_{i-1})))$  time.

For  $1 \leq i \leq m$ , we perform at most two operations in the background when we encode  $s_i$ : we dequeue a character  $a$ ; if necessary, decrement the weight of the leaf labelled  $a$ ; and re-enqueue  $a$ . These take  $O(1)$  time if we do not decrement the weight of the leaf labelled  $a$  and  $O(\log m)$  time if we do.

Suppose  $s_i$  is the first occurrence of that distinct character in  $S$ . Then the leaf  $v$  labelled  $s_i$  is inserted into  $T$  with weight  $-\lceil \log(i+n) \rceil$ . Also,  $v$ 's weight is never less than  $-\lceil \log(m+1+n) \rceil$ . Since decrementing  $v$ 's weight from  $w$  to  $w-1$  or incrementing  $v$ 's weight from  $w-1$  to  $w$  both take  $O(-w)$  time, we

spend the same amount of time decrementing  $v$ 's weight in the background as we do incrementing it in the foreground, except possibly for the time to decrease  $v$ 's weight from  $-\lceil \log(i+n) \rceil$  to  $-\lceil \log(m+1+n) \rceil$ . Thus, we spend  $O(\log^2 m)$  more time decrementing  $v$ 's weight than we do incrementing it. Since there are at most  $n$  distinct characters in  $S$ , in total, this algorithm takes

$$\sum_{i \in R} O\left(\log\left(\frac{i+n}{\#_{s_i}(s_1 \cdots s_{i-1})}\right)\right) + O(n \log^2 m)$$

time. It follows from Lemmas 1 and 2 that this is  $O((H+1)m + n \log^2 m)$ .  $\square$

## 4 Variations on Dynamic Shannon Coding

In this section, we show how to implement efficiently variations of dynamic Shannon coding for dynamic length-restricted coding, dynamic alphabetic coding and dynamic coding with unequal letter costs. Abrahams [1] surveys static algorithms for these and similar problems, but there has been relatively little work on dynamic algorithms for these problems.

We use dynamic minimax trees for length-restricted dynamic Shannon coding. For alphabetic dynamic Shannon coding, we dynamize Melhorn's version of Shannon's algorithm. For dynamic Shannon coding with unequal letter costs, we dynamize Krause's version.

### 4.1 Length-Restricted Dynamic Shannon Coding

For length-restricted coding, we are given a bound and cannot use a codeword whose length exceeds this bound. Length-restricted coding is useful, for example, for ensuring that each codeword fits in one machine word. Liddell and Moffat [12] gave a length-restricted dynamic coding algorithm that works well in practice, but it is quite complicated and they did not prove bounds on the length of the encoding it produces. We show how to length-restrict dynamic Shannon coding without significantly increasing the bound on the length of the encoding produced.

**Theorem 3.** *For any fixed integer  $\ell \geq 1$ , dynamic Shannon coding can be adapted so that it uses at most  $2\lceil \log n \rceil + \ell$  bits to encode the first occurrence of each distinct character in  $S$ , at most  $\lceil \log n \rceil + \ell$  bits to encode each remaining character in  $S$ , at most  $\left(H + 1 + \frac{1}{(2^\ell - 1) \ln 2}\right)m + O(n \log m)$  bits in total, and  $O((H+1)m + n \log^2 m)$  time.*

*Proof.* We modify the algorithm presented in Section 3 by removing the leaf labelled  $\perp$  after all of the characters in the alphabet have occurred in  $S$ , and changing how we calculate weights for the dynamic minimax tree. Whenever we would use a weight of the form  $-\lceil \log x \rceil$ , we smooth it by instead using

$$-\left\lceil \log\left(\frac{2^\ell}{(2^\ell - 1)/x + 1/n}\right) \right\rceil \geq -\min\left(\left\lceil \log\left(\frac{2^\ell x}{2^\ell - 1}\right) \right\rceil, \lceil \log n \rceil + \ell\right).$$

With these modifications, no leaf in the minimax tree is ever of depth greater than  $\lceil \log n \rceil + \ell$ . Since

$$\left\lceil \log \left( \frac{2^\ell x}{2^\ell - 1} \right) \right\rceil < \log x + 1 + \frac{\log \left( 1 + \frac{1}{2^\ell - 1} \right)^{2^\ell - 1}}{2^\ell - 1} < \log x + 1 + \frac{1}{(2^\ell - 1) \ln 2} ,$$

essentially the same analysis as for Theorem 2 shows this algorithm uses at most  $\left( H + 1 + \frac{1}{(2^\ell - 1) \ln 2} \right) m + O(n \log m)$  bits in total, and  $O((H + 1)m + n \log^2 m)$  time.  $\square$

It is straightforward to prove a similar theorem in which the number of bits used to encode  $s_i$  with  $i \in R$  is bounded above by  $\lceil \log(|\{a : a \in S\}| + 1) \rceil + \ell + 1$  instead of  $\lceil \log n \rceil + \ell$ . That is, we can make the bound in terms of the number of distinct characters in  $S$  instead of the size of the alphabet. To do this, we modify the algorithm again so that it stores a counter  $n_i$  of the number of distinct characters that have occurred in the current prefix. Whenever we would use  $n$  in a formula to calculate a weight, we use  $2(n_i + 1)$  instead.

## 4.2 Alphabetic Dynamic Shannon Coding

For alphabetic coding, the lexicographic order of the codewords must always be the same as the lexicographic order of the characters to which they are assigned. Alphabetic coding is useful, for example, because we can compare encoded strings without decoding them. Although there is an alphabetic version of minimax trees [9], it cannot be efficiently dynamized [5]. Mehlhorn [13] generalized Shannon's algorithm to obtain an algorithm for alphabetic coding. In this section, we dynamize Mehlhorn's algorithm.

**Theorem 4 (Mehlhorn, 1977).** *There exists an alphabetic prefix-free code such that, for each character  $a$  in the alphabet, the codeword for  $a$  is of length  $\lceil \log((m + n)/\#_a(S)) \rceil + 1$ .*

*Proof.* Let  $a_1, \dots, a_n$  be the characters in the alphabet in lexicographic order. For  $1 \leq i \leq n$ , let

$$f(a_i) = \frac{\#_{a_i}(S) + 1}{2(m + n)} + \sum_{j=1}^{i-1} \frac{\#_{a_j}(S) + 1}{m + n} < 1 .$$

For  $1 \leq i \neq i' \leq n$ , notice that  $|f(a_i) - f(a_{i'})| \geq \frac{\#_{a_i}(S) + 1}{2(m + n)}$ . Therefore, the first  $\left\lceil \log \left( \frac{m + n}{\#_{a_i}(S) + 1} \right) \right\rceil + 1$  bits of the binary representation of  $f(a_i)$  suffice to distinguish it. Let this sequence of bits be the codeword for  $a_i$ .  $\square$

Repeating Mehlhorn's algorithm after each character of  $S$ , as described in the introduction, is a simple algorithm for alphabetic dynamic Shannon coding. Notice that we always assign a codeword to every character in the alphabet;



thus, we do not need to prepend  $\perp$  to the current prefix of  $S$ . This algorithm uses at most  $(H + 2)m + O(n \log m)$  bits and  $O(mn)$  time to encode  $S$ .

To make this algorithm more efficient, after encoding each character of  $S$ , instead of computing an entire code-tree, we only compute the codeword for the next character in  $S$ . We use an augmented splay tree [16] to compute the necessary partial sums.

**Theorem 5.** *Alphabetic dynamic Shannon coding uses  $(H + 2)m + O(n \log m)$  bits and  $O((H + 1)m)$  time.*

*Proof.* We keep an augmented splay tree  $T$  and maintain the invariant that, after encoding  $s_1 \cdots s_{i-1}$ , there is a node  $v_a$  in  $T$  for each distinct character  $a$  in  $s_1 \cdots s_{i-1}$ . The node  $v_a$ 's key is  $a$ ; it stores  $a$ 's frequency in  $s_1 \cdots s_{i-1}$  and the sum of the frequencies of the characters in  $v_a$ 's subtree in  $T$ .

To encode  $s_i$ , we use  $T$  to compute the partial sum

$$\frac{\#_{s_i}(s_1 \cdots s_{i-1})}{2} + \sum_{a_j < s_i} \#_{a_j}(s_1 \cdots s_{i-1}),$$

where  $a_j < s_i$  means that  $a_j$  is lexicographically less than  $s_i$ . From this, we compute the codeword for  $s_i$ , that is, the first  $\left\lceil \log \left( \frac{i-1+n}{\#_{s_i}(s_1 \cdots s_{i-1})+1} \right) \right\rceil + 1$  bits of the binary representation of

$$\frac{\#_{s_i}(s_1 \cdots s_{i-1}) + 1}{2(i-1+n)} + \sum_{a_j < s_i} \frac{\#_{a_j}(s_1 \cdots s_{i-1}) + 1}{i-1+n}.$$

If  $s_i$  is the first occurrence of that character in  $S$  (i.e.,  $i \in \{1, \dots, m\} - R$ ), then we insert a node  $v_{s_i}$  into  $T$ . In both cases, we update the information stored at the ancestors of  $v_{s_i}$  and splay  $v_{s_i}$  to the root.

Essentially the same analysis as for Theorem 2 shows this algorithm uses at most  $(H + 2)m + O(n \log m)$  bits. By the Static Optimality theorem [16], it uses  $O((H + 1)m)$  time.  $\square$

### 4.3 Dynamic Shannon Coding with Unequal Letter Costs

It may be that one code letter costs more than another. For example, sending a dash by telegraph takes longer than sending a dot. Shannon [15] proved a lower bound of  $Hm \ln(2)/C$  for all algorithms, whether prefix-free or not, where the *channel capacity*  $C$  is the largest real root of  $e^{-\text{cost}(0) \cdot x} + e^{-\text{cost}(1) \cdot x} = 1$  and  $e \approx 2.71$  is the base of the natural logarithm. Krause [11] generalized Shannon's algorithm for the case with unequal positive letter costs. In this section, we dynamize Krause's algorithm.

**Theorem 6 (Krause, 1962).** *Suppose  $\text{cost}(0)$  and  $\text{cost}(1)$  are constants with  $0 < \text{cost}(0) \leq \text{cost}(1)$ . Then there exists a prefix-free code such that, for each character  $a$  in the alphabet, the codeword for  $a$  has cost less than  $\frac{\ln(m/\#_a(S))}{C} + \text{cost}(1)$ .*

*Proof.* Let  $a_1, \dots, a_k$  be the characters in  $S$  in non-increasing order by frequency. For  $1 \leq i \leq k$ , let

$$f(a_i) = \sum_{j=1}^{i-1} \frac{\#_{a_j}(S)}{m} < 1.$$

Let  $b(a_i)$  be the following binary string, where  $x_0 = 0$  and  $y_0 = 1$ : For  $j \geq 1$ , if  $f(a_i)$  is in the first  $e^{-\text{cost}(0) \cdot C}$  fraction of the interval  $[x_{j-1}, y_{j-1}]$ , then the  $j$ th bit of  $b(a_i)$  is 0 and  $x_j$  and  $y_j$  are such that  $[x_j, y_j]$  is the first  $e^{-\text{cost}(0) \cdot C}$  fraction of  $[x_{j-1}, y_{j-1}]$ . Otherwise, the  $j$ th bit of  $b(a_i)$  is 1 and  $x_j$  and  $y_j$  are such that  $[x_j, y_j]$  is the last  $e^{-\text{cost}(1) \cdot C}$  fraction of  $[x_{j-1}, y_{j-1}]$ . Notice that the cost to encode the  $j$ th bit of  $b(a_i)$  is exactly  $\frac{\ln((y_{j-1} - x_{j-1}) / (y_j - x_j))}{C}$ ; it follows that the total cost to encode the first  $j$  bits of  $b(a_i)$  is  $\frac{\ln(1/(y_j - x_j))}{C}$ .

For  $1 \leq i \neq i' \leq k$ , notice that  $|f(a_i) - f(a_{i'})| \geq \#_{a_i}(S)/m$ . Therefore, if  $y_j - x_j < \#_{a_i}(S)/m$ , then the first  $j$  bits of  $b(a_i)$  suffice to distinguish it. So the shortest prefix of  $b(a_i)$  that suffices to distinguish  $b(a_i)$  has cost less than  $\frac{\ln(e^{\text{cost}(1) \cdot C} m / \#_{a_i}(S))}{C} = \frac{\ln(m / \#_{a_i}(S))}{C} + \text{cost}(1)$ . Let this sequence of bits be the codeword for  $a_i$ .  $\square$

Repeating Krause's algorithm after each character of  $S$ , as described in the introduction, is a simple algorithm for dynamic Shannon coding with unequal letter costs. This algorithm produces an encoding of  $S$  with cost at most  $(\frac{H \ln 2}{C} + \text{cost}(1))m + O(n \log m)$  in  $O(mn)$  time.

As in Subsection 4.2, we can make this simple algorithm more efficient by only computing the codewords we need, using an augmented dynamic binary search tree. However, instead of lexicographic order, we want to keep the characters in non-increasing order by frequency in the current prefix. Since this order may change as we encode, we cannot use the Static Optimality theorem again: we use an augmented AVL tree [2] instead of a splay tree.

**Theorem 7.** *Suppose  $\text{cost}(0)$  and  $\text{cost}(1)$  are constants with  $0 < \text{cost}(0) \leq \text{cost}(1)$ . Then dynamic Shannon coding produces an encoding of  $S$  with cost at most  $(\frac{H \ln 2}{C} + \text{cost}(1))m + O(n \log m)$  in  $O(m \log n)$  time.*

*Proof.* We keep an augmented AVL tree  $T$  and maintain the invariant that, after encoding  $s_1 \dots s_{i-1}$ , there is a node  $v_a$  in  $T$  for each distinct character  $a$  in  $\perp s_1 \dots s_{i-1}$ . The node  $v_a$ 's key is  $a$ 's frequency in  $s_1 \dots s_{i-1}$ ; it stores  $a$  and the sum of the frequencies of the characters in  $v_a$ 's subtree.

If  $s_i$  occurs in  $s_1 \dots s_{i-1}$ , then we compute the sum of the frequencies of characters stored to the left of  $v_{s_i}$  in  $T$ . From this, we compute the codeword for  $s_i$  (i.e., the codeword that Krause's algorithm on  $\perp s_1 \dots s_{i-1}$  would assign to  $s_i$ ). Finally, we delete  $v_{s_i}$  and re-insert it with frequency 1 greater. In total, this takes  $O(\log n)$  time.

If  $s_i$  is the first occurrence of that character in  $S$  (i.e.,  $i \in \{1, \dots, m\} - R$ ), then we compute the codeword for  $\perp$  as described above and encode  $s_i$  as the codeword assigned to  $\perp$  followed by the binary representation of  $s_i$ 's index in the

alphabet. This encoding of  $s_i$  has cost  $O(\text{cost}(1) \cdot (\log m + \log n)) \subseteq O(\log m)$ . Then, we insert a node  $v_{s_i}$  into  $T$ . In total, this takes  $O(\log n)$  time.

Essentially the same analysis as for Theorem 2 shows this algorithm produces an encoding of  $S$  with cost at most  $(\frac{H \ln 2}{C} + \text{cost}(1))m + O(n \log m)$ .  $\square$

**Acknowledgments.** Many thanks to Faith Ellen Fich, who supervised this research, and to Julia Abrahams, Will Evans, Mordecai Golin, Charles Rackoff, Ken Sevcik and an anonymous referee. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

## References

1. J. Abrahams. Code and parse trees for lossless source encoding. *Communications in Information and Systems*, 1:113–146, 2001.
2. G.M. Adelson-Velsky and E.M. Landis. An algorithm for the organization of information. *Soviet Mathematics*, 3:1259–1263, 1962.
3. R. De Prisco and A. De Santis. On the redundancy achieved by Huffman codes. *Information Sciences*, 88:131–148, 1996.
4. N. Faller. An adaptive system for data compression. In *Proceedings of the 7th Asilomar Conference on Circuits, Systems, and Computers*, pages 593–597, 1973.
5. T. Gagie. Dynamic length-restricted coding. Master’s thesis, University of Toronto, 2003.
6. R. G. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, 24:668–674, 1978.
7. M. Golumbic. Combinatorial merging. *IEEE Transactions on Computers*, 24:1164–1167, 1976.
8. D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IERE*, 40:1098–1101, 1952.
9. D. G. Kirkpatrick and M. M. Klawe. Alphabetic minimax trees. *SIAM Journal on Computing*, 14:514–526, 1985.
10. D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985.
11. R. M. Krause. Channels which transmit letters of unequal duration. *Information and Control*, 5:13–24, 1962.
12. M. Liddell and A. Moffat. Length-restricted coding in static and dynamic frameworks. In *Proceedings of the IEEE Data Compression Conference*, pages 133–142, 2001.
13. K. Mehlhorn. A best possible bound for the weighted path length of binary search trees. *SIAM Journal on Computing*, 6:235–239, 1977.
14. R. L. Milidiú, E. S. Laber, and A. A. Pessoa. Bounding the compression loss of the FGK algorithm. *Journal of Algorithms*, 32:195–211, 1999.
15. C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.
16. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.
17. J. S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 34:825–845, 1987.

# Fractional Covering with Upper Bounds on the Variables: Solving LPs with Negative Entries

Naveen Garg and Rohit Khandekar\*

Indian Institute of Technology Delhi  
{naveen, rohitk}@cse.iitd.ernet.in

**Abstract.** We present a Lagrangian relaxation technique to solve a class of linear programs with negative coefficients in the objective function and the constraints. We apply this technique to solve (the dual of) covering linear programs with upper bounds on the variables:  $\min\{c^\top x \mid Ax \geq b, x \leq u, x \geq 0\}$  where  $c, u \in \mathbb{R}_+^m, b \in \mathbb{R}_+^n$  and  $A \in \mathbb{R}_+^{n \times m}$  have non-negative entries. We obtain a *strictly* feasible,  $(1 + \epsilon)$ -approximate solution by making  $O(m\epsilon^{-2} \log m + \min\{n, \log \log C\})$  calls to an oracle that finds the *most-violated constraint*. Here  $C$  is the largest entry in  $c$  or  $u$ ,  $m$  is the number of variables, and  $n$  is the number of covering constraints. Our algorithm follows naturally from the algorithm for the fractional packing problem and improves the previous best bound of  $O(m\epsilon^{-2} \log(mC))$  given by Fleischer [1]. Also for a fixed  $\epsilon$ , if the number of covering constraints is polynomial, our algorithm makes a number of oracle calls that is strongly polynomial.

## 1 Introduction

In last couple of decades, there has been extensive research on solving linear programs (LPs) efficiently, albeit approximately using Lagrangian relaxation. Starting from the work of Shahrokhi & Matula [2] on multicommodity flows, a series of papers including those of Plotkin et al. [3], Luby & Nisan [4], Grigoriadis & Khachiyan [5,6], Young [7,8], Garg & Könemann [9], and Fleischer [10] proposed several algorithms for packing and covering LPs. Refer to Bienstock [11] for a survey of this field. All of these algorithms rely crucially on the *non-negativity* of the coefficients in the objective function and the constraints.

In this paper we show how Lagrangian relaxation can be used to solve a class of LPs with negative coefficients in the objective function and the constraints. The basic idea behind this approach is as follows. Suppose we would like to solve an LP of the form

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{1}$$

where the entries in  $A$  and  $c$  may be negative. Suppose also that there exists an optimum solution  $x^*$  to (1) such that  $c^\top x^* \geq 0$  and  $Ax^* \geq 0$ . In such a case, we can add constraints

---

\* Supported in part by a fellowship from Infosys Technologies Ltd., Bangalore.

$c^\top x \geq 0$  and  $Ax \geq 0$  to (1) without affecting the optimum solution. The resulting linear program can then be thought of as a fractional packing problem

$$\max\{c^\top x \mid Ax \leq b, x \in P\}$$

where  $P = \{x \geq 0 \mid c^\top x \geq 0, Ax \geq 0\}$ . Such a packing problem can be solved using algorithms similar to Plotkin et al. [3], Grigoriadis & Khachiyan [12], Garg & Könemann [9], and Jansen & Zhang [13] provided one has an oracle to minimize linear functions over  $P$ .

### 1.1 Our Contribution and Previous Work

We apply the technique mentioned above to (the dual of) a covering LP with upper bounds on the variables:

$$\begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & Ax \geq b \\ & x \leq u \\ & x \geq 0 \end{aligned} \tag{2}$$

where  $c, u \in \mathbb{R}_+^m, b \in \mathbb{R}_+^n$  and  $A \in \mathbb{R}_+^{n \times m}$  have non-negative entries. For a positive integer  $k$ , let  $[k] = \{1, \dots, k\}$ . To solve (2), we assume that we have an oracle that given  $x \in \mathbb{R}_+^m$ , finds the *most-violated constraint*. Formally,

$$\text{given } x \in \mathbb{R}_+^m, \text{ the oracle finds } \min_{j \in [n]} \frac{A_j x}{b_j} \tag{3}$$

where  $A_j$  denotes the  $j$ th row of  $A$ . The dual of (2) has negative entries in the constraint matrix; we reduce it to a packing problem and prove the following theorem.

**Theorem 1.** *There exists an algorithm that given an error parameter  $\omega \in (0, 1)$ , either computes a strictly feasible,  $\exp(\omega)$ -approximation to (2), or proves that (2) is infeasible. The algorithm makes  $O(m\omega^{-2} \log m + \min\{n, \log \log C\})$  calls to the oracle (3) where  $C = \frac{\max_{i \in [m]} c_i u_i}{\min_{i: c_i u_i > 0} c_i u_i}$ .*

The LP (2) is a special case of the mixed packing and covering LP:  $\min\{c^\top x \mid Ax \geq b, Px \leq u, x \geq 0\}$  where  $c \in \mathbb{R}_+^m, b \in \mathbb{R}_+^n, u \in \mathbb{R}_+^{n'}$ ,  $A \in \mathbb{R}_+^{n \times m}$  and  $P \in \mathbb{R}_+^{n' \times m}$ . The known algorithms (Young [8]) for mixed LP however compute only *approximately feasible* solutions in which either the packing or the covering constraints are violated by a factor of  $(1 + \epsilon)$ . Our algorithm, on the other hand, computes a *strictly feasible* solution. Fleischer [1] recently presented an algorithm to compute a  $\exp(\omega)$ -approximation to (2) by making  $O(m\omega^{-2} \log(mC))$  calls<sup>1</sup> to the oracle (3). Theorem 1 thus improves her running time, especially when  $C$  is large or  $n$  is polynomial — for example, when the instance is explicitly given. We note that an approximate version of the oracle (3) that computes a row  $j' \in [n]$  such that  $A_{j'} x / b_{j'} \leq (1 + \epsilon) \min_{j \in [n]} A_j x / b_j$  is not

<sup>1</sup> Fleischer [1] defines  $C$  as  $\max_{i \in [m]} c_i / \min_{i: c_i > 0} c_i$ . However, the running time of her algorithm depends, in fact, on  $C$  as defined in Theorem 1. Note that this value of  $C$  is invariant under the scaling of variables.

strong enough to distinguish between feasibility and infeasibility of the given instance. For example, if all covering constraints are approximately satisfied by  $x = u$ , i.e.,  $Au \geq b/(1 + \epsilon)$ , the algorithm may output  $x = u$  as a solution even if it may not satisfy  $Au \geq b$  implying that the instance is infeasible. Our algorithm follows naturally from the packing algorithm; and, in fact, can be thought of as an alternate description of Fleischer's algorithm [1].

## 2 The Covering LP with Upper Bounds on the Variables

To simplify the presentation, we modify the given instance of (2) as follows. It is no loss of generality to assume that all entries of  $c$ ,  $u$ , and  $b$  are positive. This is because if  $c_i = 0$ , we can reduce the problem by setting  $x_i = u_i$  and replacing each  $b_j$  with  $b_j - A_{ji}u_i$ . If  $u_i = 0$ , then clearly  $x_i = 0$  in any feasible solution and hence we can remove this variable. Similarly, if  $b_j = 0$ , then the  $j$ th covering constraint is trivially satisfied and can be removed. We can also assume that the objective function  $c = \mathbf{1}$  is the vector of all ones; this can be achieved by working with new variables  $x'_i = c_i x_i$  and replacing each  $A_{ji}$  with  $A_{ji}/c_i$  and replacing each  $u_i$  with  $c_i u_i$ . We finally assume that  $\min_{i \in [m]} u_i = 1$ . This is achieved by dividing each entry of  $u$  and  $b$  by  $\min_{i \in [m]} u_i$ . While doing this, it is not necessary to scale the vector  $c$  by  $\min_{i \in [m]} u_i$ , since that amounts to just scaling the optimum value. Observe that after these modifications, we have  $\max_{i \in [m]} u_i = C$ . Note that the oracle (3) for the modified instance can be obtained from the given oracle for the original instance; and a solution for the original instance can be obtained from a solution of the modified instance. Since  $b$  and  $c$  have only positive entries, the optimum value of (2) is *positive*.

Before solving this linear program, we make a call to oracle (3) with  $x = u$  to find a row  $j \in [n]$  that minimizes  $A_j u / b_j$ . If  $A_j u < b_j$ , then the  $j$ th covering constraint cannot be satisfied even if we set each variable  $x_i$  to its upper bound  $u_i$ . In this case, we output “infeasible”. Therefore, in what follows, we assume that  $Au \geq b$ , i.e., the given instance is feasible.

### 2.1 Reduction to the Packing Problem

Consider the following dual LP of (2).

$$\begin{aligned} \max \quad & b^\top y - u^\top z \\ \text{s.t.} \quad & A^\top y - z \leq \mathbf{1} \\ & y, z \geq 0 \end{aligned} \tag{4}$$

Although (4) has negative entries in the constraints and the objective function, the following lemma proves that it satisfies the conditions discussed in Section 1.

**Lemma 1.** *Any optimum solution  $(y^*, z^*)$  to (4) satisfies  $b^\top y^* - u^\top z^* > 0$  and  $A^\top y^* - z^* \geq 0$ .*

*Proof.* Since the optimum value of (2) is positive, by LP duality,  $b^\top y^* - u^\top z^*$  is also positive. Now the only constraints on  $z$  are  $z_i \geq 0$  and  $z_i \geq (A^\top y)_i - 1$ . Once  $y = y^*$  is fixed, in order to maximize the objective function, we would set  $z_i$  to the smallest possible value, i.e.,  $\max\{0, (A^\top y)_i - 1\}$  and we have  $(A^\top y^*)_i - z_i^* \geq 0$ .

Thus we can add constraints  $b^\top y - u^\top z > 0$  and  $A^\top y - z \geq 0$  to (4) without affecting its optimum value. The resulting LP is

$$\begin{aligned} \max \quad & b^\top y - u^\top z \\ \text{s.t.} \quad & A^\top y - z \leq \mathbf{1} \\ & b^\top y - u^\top z > 0 \\ & A^\top y - z \geq 0 \\ & y, z \geq 0 \end{aligned} \quad (5)$$

We now define an instance of the packing problem. Define a polytope  $P \subset \mathbb{R}_+^{n+m}$  and functions  $f_i : P \rightarrow \mathbb{R}_+, i \in [m]$  as follows.

$$\begin{aligned} P = \{ & (y, z) \geq 0 \mid A^\top y - z \geq 0, b^\top y - u^\top z = 1 \}, \\ f_i(y, z) = & (A^\top y)_i - z_i \text{ for } i \in [m]. \end{aligned} \quad (6)$$

Now consider the packing problem

$$\min_{(y,z) \in P} \max_{i \in [m]} f_i(y, z). \quad (7)$$

It is easy to see that (5) has an optimum value  $\lambda$  if and only if (7) has an optimum value  $1/\lambda$ . Furthermore, an  $\exp(\omega)$ -approximate solution of (7) can be scaled to obtain an  $\exp(\omega)$ -approximate solution of (5). It is therefore sufficient to obtain an  $\exp(\omega)$  approximation to (7).

### 3 The Packing Algorithm

For completeness, we outline an algorithm for the packing problem. Given a convex set  $P \subseteq \mathbb{R}^n$  and  $m$  convex non-negative functions  $f_i : P \rightarrow \mathbb{R}_+, i \in [m]$ , the packing problem is to compute

$$\lambda^* = \min_{y \in P} \lambda(y) \quad \text{where} \quad \lambda(y) := \max_{i \in [m]} f_i(y).$$

We refer to this problem as **PRIMAL**. For a non-zero vector  $x \in \mathbb{R}_+^m$ , define  $\underline{x} := x / \sum_{i=1}^m x_i$  as the normalized vector and define  $d(x) := \min_{y \in P} \langle \underline{x}, f(y) \rangle$  where  $\langle \underline{x}, f(y) \rangle = \sum_i \underline{x}_i f_i(y)$  denotes the inner product. It is easy to see that  $d(x) \leq \lambda^* \leq \lambda(y)$  for any  $x > 0$  and  $y \in P$ . We refer to the problem of finding  $x > 0$  such that  $d(x)$  is maximum as the **DUAL**. The duality theorem (see, e.g., Rockafellar [14], p.393) implies that

$$\min_{y \in P} \lambda(y) = \max_{x > 0} d(x).$$

The algorithm assumes an oracle that given  $x > 0$ , computes  $y \in P$  such that

$$\langle \underline{x}, f(y) \rangle \leq \exp(\epsilon) \cdot d(x) \quad (8)$$

for a constant  $\epsilon \in (0, 1)$ .

**Theorem 2.** *The algorithm in Figure 1 computes  $x^* > 0$  and  $y^* \in P$  such that  $\lambda(y^*) \leq \exp(3\epsilon) \cdot d(x^*)$ . The duality then implies that  $x^*$  and  $y^*$  form the near-optimum solutions to the **DUAL** and **PRIMAL** problems respectively. The algorithm makes  $O(m\epsilon^{-2} \log m)$  calls to the oracle (8).*

For the proof of Theorem 2, we refer to [15].

(1)	$x := \mathbf{1}$	{ $\mathbf{1}$ is a vector of all ones}
(2)	Find $y \in P$ such that $\langle \underline{x}, f(y) \rangle \leq \exp(\epsilon) \cdot d(x)$	{oracle call}
(3)	$x^* := x, d^* = \langle \underline{x}, f(y) \rangle$	{current dual and its value}
(4)	$r := 0$	{initialize the round number}
(5)	Repeat	
(6)	$r := r + 1$	{round $r$ begins}
(7)	Find $y_r \in P$ such that $\langle \underline{x}, f(y_r) \rangle \leq \exp(\epsilon) \cdot d(x)$	{oracle call}
(8)	$w_r := 1 / \max_{i \in [m]} f_i(y_r)$	{pick $y_r$ to an extent $w_r$ }
(9)	For $i \in [m]$ do $x_i := x_i \cdot \exp(\epsilon w_r f_i(y_r))$	{update the dual variables}
(10)	If $\langle \underline{x}, f(y_r) \rangle > d^*$ then $x^* := x, d^* := \langle \underline{x}, f(y_r) \rangle$	{keep track of the best dual}
(11)	Until $(\max_{i \in [m]} x_i \geq m^{2/\epsilon})$	{round $r$ ends}
(12)	Output $x^*$ and $y^* = (\sum_{s=1}^r w_s y_s) / (\sum_{s=1}^r w_s)$	

**Fig. 1.** The packing algorithm with exponential updates of the duals

## 4 Implementing the Desired Oracle

To solve (7) using the packing algorithm, we need an oracle that given a non-zero vector  $x \in \mathbb{R}_+^m$ , computes  $(y, z) \in P$  that minimizes  $\underline{x}^\top (A^\top y - z)$  within a factor of  $\exp(\epsilon)$ , where  $\underline{x} = x / \sum_{i=1}^m x_i$ . Since  $\sum_{i=1}^m x_i$  is fixed, we need to approximate

$$R(x) := \min_{(y,z) \in P} x^\top (A^\top y - z). \quad (9)$$

Since  $b^\top y - u^\top z = 1$  for  $(y, z) \in P$ , we have

$$R(x) = \min \left\{ \frac{x^\top (A^\top y - z)}{b^\top y - u^\top z} \mid y, z \geq 0, A^\top y - z \geq 0, b^\top y - u^\top z > 0 \right\}.$$

We now show how to find  $(y, z)$  that achieves an approximate value of  $R(x)$ .

For  $x \in \mathbb{R}_+^m$  and  $\alpha \geq 0$ , define  $(x|_\alpha) \in \mathbb{R}_+^m$  as the “truncated” vector with  $(x|_\alpha)_i = \min\{x_i, \alpha u_i\}$  for  $i \in [m]$ . Define

$$r_\alpha(x) = \min_{j \in [n]} \frac{A_j(x|_\alpha)}{b_j}$$

where  $A_j$  is the  $j$ th row of  $A$ . Given  $x$  and  $\alpha$ , we can compute  $r_\alpha(x)$  by making one call to oracle (3). The proof of the following lemma is omitted due to lack of space.

**Lemma 2.** For a fixed  $x \in \mathbb{R}_+^m$ , the function  $r_\alpha(x)$  is a non-decreasing and concave function of  $\alpha$ .

**Lemma 3.** For any  $x \in \mathbb{R}_+^m$  with  $x_i > 0$  for  $i \in [m]$ , we have  $r_{R(x)}(x) = R(x)$ .

*Proof.* Let  $y, z \geq 0$  be such that  $A^\top y - z \geq 0, b^\top y - u^\top z > 0$  and

$$R(x) = \frac{x^\top (A^\top y - z)}{b^\top y - u^\top z}.$$

It is no loss of generality to assume that the above minimum is achieved for  $\langle \mathbf{1}, y \rangle = 1$ ; this can be achieved by scaling  $y, z$ . Recall that  $R(x)$  is the minimum possible value of this fraction.



*Claim.* Without loss of generality, we can assume that the vector  $z$  is of the form

$$z_i = \begin{cases} (A^\top y)_i, & \text{if } x_i > R(x) \cdot u_i; \\ 0, & \text{otherwise.} \end{cases} \quad (10)$$

*Proof.* Note that the coefficient of  $z_i$  in the numerator is  $x_i$ , while its coefficient in the denominator is  $u_i$ . If  $x_i/u_i \leq R(x)$ , we can set  $z_i = 0$ , since in this case, by decreasing  $z_i$ , the value of the fraction does not increase. If on the other hand  $x_i/u_i > R(x)$ , the variable  $z_i$  must be set to its highest possible value,  $(A^\top y)_i$ , since otherwise by increasing  $z_i$ , we can decrease the value of the fraction. On increasing  $z_i$ , however, the value of the denominator  $b^\top y - u^\top z$  may reach zero. But in this case, since  $x_i/u_i > R(x)$ , the numerator  $x^\top (A^\top y - z)$  reaches zero “before” the denominator. Thus we would have  $x^\top (A^\top y - z) = 0$  and  $b^\top y - u^\top z > 0$  for some  $y, z \geq 0$ . Since  $x_i > 0$  for  $i \in [m]$ , it must be that  $A^\top y - z = 0$ . This, in turn, implies that (4) is unbounded and (2) is infeasible yielding a contradiction. This proves the claim.

Now since  $x^\top (A^\top y - z) = R(x) \cdot (b^\top y - u^\top z)$ , we have  $x^\top A^\top y - (x^\top - R(x) \cdot u^\top)z = R(x) \cdot b^\top y$ .

**Lemma 4.** *If  $z$  satisfies  $z_i = (A^\top y)_i$  if  $x_i > \alpha \cdot u_i$ , and 0 otherwise for some  $\alpha$ , then we have  $x^\top A^\top y - (x^\top - \alpha \cdot u^\top)z = (x|_\alpha)^\top A^\top y$ .*

*Proof.* We show that  $x_i(A^\top y)_i - (x_i - \alpha \cdot u_i)z_i = (x|_\alpha)_i(A^\top y)_i$  holds for each  $i \in [m]$ . If  $x_i - \alpha \cdot u_i \leq 0$ , we have  $z_i = 0$  and  $(x|_\alpha)_i = x_i$ , hence the equality holds. On the other hand, if  $x_i - \alpha \cdot u_i > 0$ , we have  $z_i = (A^\top y)_i$  and  $x_i(A^\top y)_i - (x_i - \alpha \cdot u_i)z_i = (\alpha \cdot u_i)(A^\top y)_i = (x|_\alpha)_i(A^\top y)_i$ . This proves Lemma 4.

Thus we have  $(x|_{R(x)})^\top A^\top y = R(x) \cdot b^\top y$ , i.e.,

$$R(x) = \frac{(x|_{R(x)})^\top A^\top y}{b^\top y}.$$

Further we can assume that  $y$  is of the form

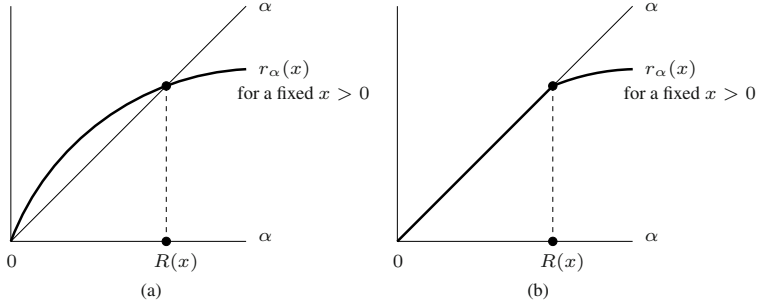
$$\begin{aligned} y_j &= 1 && \text{for a row } j \in [n] \text{ such that } ((x|_{R(x)})^\top A^\top)_j / b_j \text{ is minimum,} \\ y_{j'} &= 0 && \text{for } j' \neq j. \end{aligned} \quad (11)$$

This is true because by replacing  $y$  with the vector as defined above, the value of the ratio does not increase. Therefore indeed we have  $r_{R(x)}(x) = \min_j A_j(x|_{R(x)})/b_j = R(x)$  and the proof of Lemma 3 is complete.

**Lemma 5.** *For any  $x \in \mathbb{R}_+^m$  with  $x_i > 0$  for  $i \in [m]$ , we have  $r_\alpha(x) < \alpha$  if and only if  $\alpha > R(x)$ .*

*Proof.* Since  $r_\alpha$  is a concave function of  $\alpha$  and  $r_0(x) = 0, r_{R(x)} = R(x)$ , we have  $r_\alpha(x) \geq \alpha$  for  $0 \leq \alpha \leq R(x)$ . Therefore  $r_\alpha(x) < \alpha$  implies  $\alpha > R(x)$ .

Now let  $\alpha > R(x)$ . Again using the concavity of  $r_\alpha$ , we have  $r_\alpha(x) \leq \alpha$ . We now show that strict inequality holds. Let  $(y, z)$  be vectors as given in (11) and (10) that achieve the minimum ratio  $R(x)$ . Note that we have  $(A^\top y)_i - z_i > 0$  for some  $i \in [m]$ ;



**Fig. 2.** The possible forms of the function  $r_\alpha$  for a fixed  $x > 0$ . (a) case if  $\min_{j \in [n]} A_j u/b_j > 1$ , (b) case if  $\min_{j \in [n]} A_j u/b_j = 1$ .

otherwise the argument in the proof of Lemma 3 implies that (2) is infeasible. For such  $i \in [m]$ , we have  $(A^\top y)_i > 0$  and  $z_i = 0$ . This implies  $x_i \leq R(x) \cdot u_i$  and  $A_{ij} > 0$  for index  $j \in [n]$  defined in (11). Note that  $x_i \leq R(x) \cdot u_i < \alpha \cdot u_i$  implies  $(x|_\alpha)_i = (x|_{R(x)})_i = x_i > 0$ . This combined with  $A_{ij} > 0$  and  $(x|_\alpha)_i \leq \alpha/R(x) \cdot (x|_{R(x)})_i$  implies that  $A_j(x|_\alpha)/b_j < \alpha/R(x) \cdot A_j(x|_{R(x)})/b_j = \alpha/R(x) \cdot R(x) = \alpha$ . Since this holds for all  $j \in [n]$  that attain the minimum in (11), we have  $r_\alpha(x) < \alpha$ . Thus Lemma 5 is proved.

Now from Lemmas 2,3, and 5, the function  $r_\alpha$  must be of the form given in Figure 2. Fix an  $x > 0$ . Consider an  $\alpha \in [0, \min_{i \in [m]} x_i/u_i]$ . Note that  $(x|_\alpha) = \alpha \cdot u$ . Therefore we have  $r_\alpha(x) = \alpha \cdot \min_{j \in [n]} A_j u/b_j$ . Since (2) is feasible, we have  $\min_{j \in [n]} A_j u/b_j \geq 1$ . Now if  $\min_{j \in [n]} A_j u/b_j > 1$ , we have  $r_\alpha(x) > \alpha$ . In this case, from Lemmas 2,3, and 5, we get that  $\alpha = 0$  and  $\alpha = R(x)$  are the only values of  $\alpha$  satisfying  $r_\alpha(x) = \alpha$  and the function  $r_\alpha$  has the form given in Figure 2 (a). On the other hand, if  $\min_{j \in [n]} A_j u/b_j = 1$ , we have  $r_\alpha(x) = \alpha$ . In this case, the function  $r_\alpha$  has the form given in Figure 2 (b). In either case,  $R(x)$  is the largest value of  $\alpha$  such that  $r_\alpha(x) = \alpha$ .

#### 4.1 Computing $R(1)$

In the packing algorithm (Figure 1), we initialize  $x = 1$ . In this section, we describe how to compute  $R(1)$  and prove the following lemma.

**Lemma 6.** *The exact value of  $R(1)$  can be computed by making  $O(n + \log m)$  calls to the oracle (3). Furthermore, given  $\epsilon \in (0, 1)$ , an  $\exp(\epsilon)$  approximation to  $R(1)$  can be computed by making  $O(\log \log C + \log(1/\epsilon))$  calls to the oracle (3).*

**Lemma 7.**  *$R(1) \geq 1/\max_i u_i = 1/C$ . Further if  $R(1) \geq 1/\min_i u_i = 1$ , then  $R(1) = \min_{j \in [n]} A_j 1/b_j$ .*

*Proof.* We first argue that  $R(u) \geq 1$ . Recall that

$$R(u) = \min_{(y,z)} \frac{u^\top A^\top y - u^\top z}{b^\top y - u^\top z}$$

where the minimum is taken over  $(y, z) \geq 0$  such that  $u^\top A^\top y - u^\top z \geq 0$  and  $b^\top y - u^\top z > 0$ . Since (2) is feasible, we have  $\min_{j \in [n]} A_j u / b_j \geq 1$ . Hence to minimize the above ratio, one has to set  $z$  to zero and this gives  $R(u) = \min_{j \in [n]} A_j u / b_j \geq 1$ . Since  $C \cdot \mathbf{1} \geq u$ , we have  $R(C \cdot \mathbf{1}) \geq R(u) \geq 1$ . Therefore  $R(\mathbf{1}) = R(C \cdot \mathbf{1}) / C \geq 1/C$ .

Now assume  $R(\mathbf{1}) \geq 1$ . Since  $u \geq \mathbf{1}$ , we have  $\min\{1, R(\mathbf{1}) \cdot u_i\} = 1$  for each  $i \in [m]$ . Therefore  $(\mathbf{1}|_{R(\mathbf{1})}) = \mathbf{1}$ . From Lemma 3, we have  $R(\mathbf{1}) = r_{R(\mathbf{1})}(\mathbf{1})$ . Therefore,  $R(\mathbf{1}) = \min_{j \in [n]} A_j \mathbf{1} / b_j$  as desired.

**Computing  $R(\mathbf{1})$  approximately.** Now we describe a simple procedure to compute an  $\exp(\epsilon)$  approximation to  $R(\mathbf{1})$  where  $\epsilon \in (0, 1)$ . We first compute  $r_1(\mathbf{1})$ . If  $r_1(\mathbf{1}) \geq 1$ , Lemma 5 implies that  $R(\mathbf{1}) \geq 1$ . In this case, Lemma 7 gives that  $R(\mathbf{1}) = \min_{j \in [n]} A_j \mathbf{1} / b_j$  and we can compute this value by making one more oracle call.

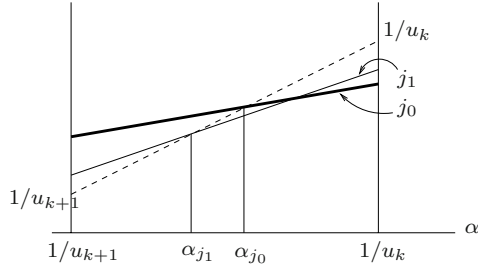
On the other hand, if  $r_1(\mathbf{1}) < 1$ , then we have  $1/C \leq R(\mathbf{1}) < 1$ . In such a case, we have lower and upper bounds on  $R(\mathbf{1})$  within a factor  $C$ . We do a bisection search as follows. We set  $\alpha = \sqrt{1/C}$  and compute  $r_\alpha(\mathbf{1})$ . If  $r_\alpha(\mathbf{1}) \geq \alpha$ , Lemma 5 implies that  $R(\mathbf{1}) \in [\alpha, 1]$ . On the other hand, if  $r_\alpha(\mathbf{1}) < \alpha$ , we get that  $R(\mathbf{1}) \in [1/C, \alpha]$ . In either case, we reduce the ratio of upper and lower bounds from  $C$  to  $\sqrt{C}$ . By repeating this  $O(\log \log C + \log(1/\epsilon))$  times, we can reduce the ratio of upper and lower bounds to  $\exp(\epsilon)$ . Thus we can compute  $\alpha$  such that  $\exp(-\epsilon) \cdot \alpha \leq R(\mathbf{1}) < \alpha$ .

**Computing  $R(\mathbf{1})$  exactly.** We use the fact that  $R(\mathbf{1})$  is the maximum value of  $\alpha$  such that  $r_\alpha(\mathbf{1}) = \alpha$  (see Figure 2) to compute the exact value of  $R(\mathbf{1})$ . Reorder the indices  $i \in [m]$  such that  $1 = u_1 \leq u_2 \leq \dots \leq u_m = C$ . We first compute  $r_1(\mathbf{1})$ . If  $r_1(\mathbf{1}) \geq 1/u_1 = 1$ , then Lemmas 5 and 7 imply that  $R(\mathbf{1}) = \min_{j \in [n]} A_j \mathbf{1} / b_j$  and we can compute it exactly. We therefore assume that  $1/u_m \leq R(\mathbf{1}) < 1/u_1$ . Given  $i \in [m]$ , we can determine if  $R(\mathbf{1}) \leq 1/u_i$ . This is done by computing  $r_\alpha(\mathbf{1})$  with  $\alpha = 1/u_i$  and comparing its value with  $\alpha$  (see Lemma 5). Therefore, by making  $O(\log m)$  oracle calls, we can compute (unique)  $k \in [m - 1]$  such that  $1/u_{k+1} \leq R(\mathbf{1}) < 1/u_k$ . Now note that for  $\alpha \in [1/u_{k+1}, 1/u_k]$ , we have

$$(\mathbf{1}|_\alpha)_i = \min\{1, \alpha u_i\} = \begin{cases} \alpha u_i, & \text{for } i = 1, \dots, k; \\ 1, & \text{for } i = k + 1, \dots, m. \end{cases}$$

Therefore for any row  $j \in [n]$ , we have that  $A_j(\mathbf{1}|_\alpha)/b_j$  is a linear function of  $\alpha$  in the range  $\alpha \in [1/u_{k+1}, 1/u_k]$ . In Figure 3, we have shown the linear functions corresponding to some rows  $j_0$  and  $j_1$ . Recall that the function  $r_\alpha(\mathbf{1})$  is the “lower envelope” of these functions and our objective is to compute  $R(\mathbf{1})$  which is the largest value of  $\alpha$  such that  $r_\alpha(\mathbf{1}) = \alpha$ . Thus  $R(\mathbf{1})$  is the point at which the lower envelope intersects the line corresponding to the linear function  $\alpha$  which is shown dotted in Figure 3.

We first make an oracle call to compute  $j_0 = \arg \min_{j \in [n]} A_j(\mathbf{1}|_\alpha)/b_j$  where  $\alpha = 1/u_k$ . (Refer to Figure 3.) The solid line corresponds to the linear function  $A_{j_0}(\mathbf{1}|_\alpha)/b_{j_0}$ . We assume that the oracle also returns the most-violated constraint. Since we know the row  $A_{j_0}$ , we can compute  $\alpha_{j_0}$  which is the value of  $\alpha$  for which  $A_{j_0}(\mathbf{1}|_\alpha)/b_{j_0} = \alpha$ . As shown in the figure,  $\alpha_{j_0}$  is the value of  $\alpha$  at which the dotted and the solid lines intersect.



**Fig. 3.** Eliminating rows one-by-one to compute  $R(\mathbf{1})$  exactly

We then make another oracle call to compute  $j_1 = \arg \min_{j \in [n]} A_j(\mathbf{1}|_{\alpha_{j_0}})/b_j$ . The light line in Figure 3 corresponds to the linear function  $A_{j_1}(\mathbf{1}|_{\alpha})/b_{j_1}$ . If  $j_1 = j_0$ , then since the solid line goes below the dotted line for  $\alpha > \alpha_{j_0}$ , we know that  $\alpha_{j_0}$  is the largest value of  $\alpha$  such that  $r_\alpha(\mathbf{1}) = \alpha$  and  $R(\mathbf{1}) = \alpha_0$ . On the other hand, if  $j_1 \neq j_0$ , we find  $\alpha_{j_1}$  which is the value of  $\alpha$  at which the dotted and the light lines intersect. We continue this to compute rows  $j_2, j_3, \dots$  till we find  $j_{l+1} = j_l$  for some  $l \geq 0$ . Since  $R(\mathbf{1})$  itself is a value of  $\alpha$  that corresponds to the intersection of the dotted line and the line corresponding to the function  $A_{j'}(\mathbf{1}|_{\alpha})/b_{j'}$  for some row  $j' \in [n]$ , after  $O(n)$  such iterations, we find  $j_{l+1} = j_l = j'$  for some  $l \geq 0$ . Thus we can compute  $R(\mathbf{1})$  by making  $O(n + \log m)$  calls to the oracle (3).

**Computing  $(y, z) \in P$  that approximates  $R(\mathbf{1})$  well.** Given an approximate or an exact value of  $R(\mathbf{1})$ , we still need to find  $(y, z) \in P$  that achieves an approximation to  $R(\mathbf{1})$ .

**Lemma 8.** *Given  $x \in \mathbb{R}_+^m$  such that  $x_i > 0$  for  $i \in [m]$  and  $\alpha$  such that  $\exp(-\epsilon) \cdot \alpha \leq R(x) < \alpha$ , we can compute  $(y, z) \in P$  such that  $x^\top(A^\top y - z) \leq \exp(\epsilon) \cdot R(x)$  by making one call to the oracle (3).*

*Proof.* We use an argument similar to the one given in the proof of Lemma 3. We compute  $j \in [n]$  such that  $A_j(x|_\alpha)/b_j$  is minimum and set  $y_j = 1$  and  $y_{j'} = 0$  for  $j' \neq j$ . Therefore we have  $(x|_\alpha)^\top A^\top y = r_\alpha(x) \cdot b^\top y$ . We also set

$$z_i = \begin{cases} (A^\top y)_i, & \text{if } x_i > \alpha \cdot u_i; \\ 0, & \text{otherwise.} \end{cases}$$

Now Lemma 4 implies that  $x^\top A^\top y - (x^\top - \alpha \cdot u^\top)z = (x|_\alpha)^\top A^\top y$ . Hence  $x^\top A^\top y - (x^\top - \alpha \cdot u^\top)z = r_\alpha(x) \cdot b^\top y$ . This, in turn, implies  $x^\top(A^\top y - z) = r_\alpha(x) \cdot b^\top y - \alpha \cdot u^\top z < \alpha(b^\top y - u^\top z)$ . The strict inequality follows from  $r_\alpha(x) < \alpha$  (Lemma 5) and  $b^\top y > 0$ . Therefore we have  $b^\top y - u^\top z > 0$  and

$$\frac{x^\top(A^\top y - z)}{b^\top y - u^\top z} < \alpha \leq \exp(\epsilon) \cdot R(x).$$

Thus the proof is complete.

Using the above lemma, we can compute  $(y, z) \in P$  that achieves an  $\exp(\epsilon)$  approximation to  $R(\mathbf{1})$ .

## 5 Algorithm for Solving (7)

In the packing algorithm (Figure 1), we initialize  $x = \mathbf{1}$ . We set  $\epsilon = \omega/4$  where  $\omega \in (0, 1)$  is the given error parameter. As explained in the previous section, we compute an  $\alpha$  such that  $\exp(-\epsilon) \cdot \alpha \leq R(x) < \alpha$ . Using Lemma 8, we compute  $(y, z) \in P$  that forms a  $\exp(\epsilon)$  approximation to the oracle call and update the “dual variables”  $x$  (step (9) in Figure 1). During the algorithm, the values of  $x_i$  for  $i \in [m]$  never decrease. Therefore, the value of

$$R(x) = \min_{(y,z) \in P} x^\top (A^\top y - z)$$

also never decreases. We repeat this till we find that  $R(x) \geq \alpha$ ; this can be detected by checking if  $r_\alpha(x) \geq \alpha$  (Lemma 5). In such a case, we update the value of  $\alpha$  as  $\alpha := \exp(\epsilon) \cdot \alpha$  and repeat.<sup>2</sup>

Let  $\alpha_0$  be the initial value of  $\alpha$ . Note that we have  $R(\mathbf{1}) \in [\alpha_0 \exp(-\epsilon), \alpha_0]$ . We say that the algorithm is in phase  $p \geq 0$  if the current value of  $\alpha$  is  $\alpha_0 \exp(p\epsilon)$ .

**Bounding the number of oracle calls.** We designate an oracle call in which we do not find the desired  $(y, z) \in P$  as “unfruitful”. After each unfruitful oracle call, we update the value of  $\alpha$  and go into the next phase. Hence the number of unfruitful calls to (3) is at most the number of phases in the algorithm. The following lemma proves a bound on the number of phases in the algorithm.

**Lemma 9.** *The algorithm has  $O(\epsilon^{-2} \log m)$  phases.*

*Proof.* The algorithm initializes  $x = \mathbf{1}$  and terminates when for some  $i \in [m]$ , we have  $x_i \geq m^{2/\epsilon}$  (step (11) in Figure 1). Thus the maximum value  $\alpha$  can take is  $\alpha_0 \cdot \exp(\epsilon) m^{2/\epsilon}$ . Since the value of  $\alpha$  increases by  $\exp(\epsilon)$  in each phase, the number of phases in the algorithm is  $O(\epsilon^{-2} \log m)$ .

Thus the number of unfruitful oracle calls is  $O(\epsilon^{-2} \log m)$ . Call an oracle call in which we find  $(y, z) \in P$  satisfying the desired inequality as “fruitful”. Clearly the number of fruitful oracle calls is equal to the number of oracle calls in the packing algorithm and from Theorem 2, this is  $O(m\epsilon^{-2} \log m)$ . In the pre-processing, to compute an approximate value of  $R(\mathbf{1})$ , we make  $O(\min\{n + \log m, \log \log C + \log(1/\epsilon)\})$  oracle calls. Thus the total number of calls to the oracle (3) in the algorithm is  $O(m\epsilon^{-2} \log m + \min\{n + \log m, \log \log C + \log(1/\epsilon)\}) = O(m\omega^{-2} \log m + \min\{n, \log \log C\})$ .

**Proving feasibility and near-optimality of the solutions.** Now from Theorem 2, the algorithm outputs  $\tilde{x} \geq 0$  and  $(\tilde{y}, \tilde{z}) \in P$  such that

$$\lambda(\tilde{y}, \tilde{z}) \leq \exp(3\epsilon) \cdot d(\tilde{x}) \quad (12)$$

where

$$\lambda(\tilde{y}, \tilde{z}) = \max_{i \in [m]} \left( (A^\top \tilde{y})_i - \tilde{z}_i \right) \quad \text{and} \quad d(\tilde{x}) = \min_{(y,z) \in P} \langle \tilde{x}, f(y, z) \rangle \quad (13)$$

<sup>2</sup> This technique of updating  $\alpha$  in multiples of  $\exp(\epsilon)$  was first used by Fleischer [10] to reduce the running time of the maximum multicommodity flow algorithm.

where  $\tilde{x} = \tilde{x} / \sum_{i=1}^m \tilde{x}_i$ . Recall that the algorithm (Figure 1) outputs the best dual solution found in any round and hence

$$\tilde{x} = x^r \text{ where } r = \arg \max_{1 \leq r \leq N} d(x^r)$$

where  $x^r$  denotes the value of  $x$  at the end of round  $r$ . Let  $p$  be the phase in which the algorithm computes the best dual solution  $\tilde{x}$ . Let  $\hat{x}$  be the value of the dual variables  $x$  at the end of the phase  $p - 1$ . Let  $\alpha_{p-1} = \alpha_0 \exp((p - 1)\epsilon)$  be the value of  $\alpha$  in phase  $p - 1$ .

**Lemma 10.** *The vectors  $x^* = (\hat{x}|_{\alpha_{p-1}})/\alpha_{p-1}$  and  $(y^*, z^*) = (\tilde{y}, \tilde{z})/\lambda(\tilde{y}, \tilde{z})$  form feasible solutions to the primal linear program (2) and the dual linear program (4) respectively. Furthermore,*

$$\mathbf{1}^\top x^* \leq \exp(\omega) \cdot (b^\top y^* - u^\top z^*).$$

Thus  $x^*$  and  $(y^*, z^*)$  form near-optimum solutions to (2) and (4) respectively.

*Proof.* Since  $(\hat{x}|_{\alpha_{p-1}}) \leq \alpha_{p-1}u$ , we have  $x^* \leq u$ . Since the oracle call with  $x = \hat{x}$  was unfruitful at the end of phase  $p - 1$ , we have  $r_{\alpha_{p-1}}(\hat{x}) \geq \alpha_{p-1}$ . Therefore  $\min_{j \in [n]} A_j(\hat{x}|_{\alpha_{p-1}})/b_j = r_{\alpha_{p-1}}(\hat{x}) \geq \alpha_{p-1}$  and hence  $\min_{j \in [n]} A_j x^*/b_j \geq 1$ . Thus  $x^*$  forms a feasible solution to (2). From the definition of  $\lambda(\tilde{y}, \tilde{z})$ , it is clear that  $A^\top y^* - z^* \leq \mathbf{1}$  so that  $(y^*, z^*)$  forms a feasible solution to (4).

Now let  $\alpha_p = \alpha_0 \exp(p\epsilon)$  be the value of  $\alpha$  in phase  $p$ . We have

$$\begin{aligned} \mathbf{1}^\top x^* &= \frac{\sum_{i=1}^m (\hat{x}|_{\alpha_{p-1}})_i}{\alpha_{p-1}} \\ &\leq \frac{\sum_{i=1}^m \tilde{x}_i}{\alpha_{p-1}} \quad (\text{since } x \text{ does not decrease, we have } \tilde{x} \geq (\hat{x}|_{\alpha_{p-1}})) \\ &= \exp(\epsilon) \cdot \frac{\sum_{i=1}^m \tilde{x}_i}{\alpha_p} \quad (\text{since } \alpha_p = \exp(\epsilon) \cdot \alpha_{p-1}) \end{aligned}$$

From the definitions (9) and (13) of  $R(x)$  and  $d(x)$  respectively, we have  $R(x) = d(x) \cdot \sum_{i=1}^m x_i$ . Since the oracle call with  $x = \tilde{x}$  was fruitful in phase  $p$ , we have  $\alpha_p \geq R(\tilde{x}) = d(\tilde{x}) \cdot \sum_{i=1}^m \tilde{x}_i$ . Therefore

$$\begin{aligned} \mathbf{1}^\top x^* &\leq \exp(\epsilon) \cdot \frac{1}{d(\tilde{x})} \\ &\leq \exp(4\epsilon) \cdot \frac{1}{\lambda(\tilde{y}, \tilde{z})} \quad (\text{from (12)}) \\ &= \exp(\omega) \cdot \frac{b^\top \tilde{y} - u^\top \tilde{z}}{\lambda(\tilde{y}, \tilde{z})} \quad (\text{since } (\tilde{y}, \tilde{z}) \in P \text{ and } \omega = 4\epsilon) \\ &= \exp(\omega) \cdot (b^\top y^* - u^\top z^*). \end{aligned}$$

Therefore the proof is complete.

Note that we can modify the algorithm to keep track of the values of the dual variables at the end of the previous round and thus be able to compute  $\hat{x}$ ,  $\alpha_{p-1}$ , and hence  $x^*$ .

Thus the proof of Theorem 1 is complete.

## 6 Conclusion

We presented a technique to handle negative entries in the objective function and the constraints via Lagrangian relaxation. We applied this technique to the fractional covering problem with upper bounds on the variables. We reduced the dual of this problem to the standard packing framework and derived an algorithm for this problem. However, the technique seems to be very limited in its applicability since it needs an optimum solution with non-negative values for the objective function and the constraints. The negative entries can also be handled by working with perturbed functions as done in [15]. However, there is still no fairly general and satisfactory way of taking care of negative entries.

## References

1. Fleischer, L.: A fast approximation scheme for fractional covering problems with variable upper bounds. In: Proceedings, ACM-SIAM Symposium on Discrete Algorithms. (2004) 994–1003
2. Shahrokhi, F., Matula, D.: The maximum concurrent flow problem. *J. ACM* **37** (1990) 318–334
3. Plotkin, S., Shmoys, D., Tardos, E.: Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.* **20** (1995) 257–301
4. Luby, M., Nisan, N.: A parallel approximation algorithm for positive linear programming. In: Proceedings, ACM Symposium on Theory of Computing. (1993) 448–457
5. Grigoriadis, M., Khachiyan, L.: Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM J. Optimization* **4** (1994) 86–107
6. Grigoriadis, M., Khachiyan, L.: Approximate minimum-cost multicommodity flows in  $\tilde{O}(\epsilon^{-2} knm)$  time. *Math. Programming* **75** (1996) 477–482
7. Young, N.: Randomized rounding without solving the linear program. In: Proceedings, ACM-SIAM Symposium on Discrete Algorithms. (1995) 170–178
8. Young, N.: Sequential and parallel algorithms for mixed packing and covering. In: Proceedings, IEEE Symposium on Foundations of Computer Science. (2001) 538–546
9. Garg, N., Könemann, J.: Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In: Proceedings, IEEE Symposium on Foundations of Computer Science. (1998) 300–309
10. Fleischer, L.: Approximating fractional multicommodity flow independent of the number of commodities. *SIAM J. Discrete Math.* **13** (2000) 505–520
11. Bienstock, D.: Potential Function Methods for Approximately Solving Linear Programming Problems: Theory and Practice. Kluwer Academic Publishers, Boston (2002) An early version also appears as CORE Lecture Series Monograph, Core, UCL, Belgium (2001) (download from [www.core.ucl.ac.be](http://www.core.ucl.ac.be)).
12. Grigoriadis, M., Khachiyan, L.: Coordination complexity of parallel price-directive decomposition. *Math. Oper. Res.* **21** (1994) 321–340
13. Jansen, K., Zhang, H.: Approximation algorithms for general packing problems with modified logarithmic potential function. In: Theoret. Comput. Sci. (2002) 255–266
14. Rockafellar, T.: *Convex Analysis*. Princeton University Press, Princeton, NJ (1970)
15. Khandekar, R.: Lagrangian relaxation based algorithms for convex programming problems. PhD thesis, Indian Institute of Technology Delhi (2004) Available at <http://www.cse.iitd.ernet.in/~rohitk>.

# Negotiation-Range Mechanisms: Coalition-Resistant Markets

Rica Gonen

The Hebrew University of Jerusalem, School of Computer Science and Engineering,  
Jerusalem, Israel

**Abstract.** Negotiation-range mechanisms offer a novel approach to achieving efficient markets based on finding the maximum weighted matching in a weighted bipartite graph connecting buyers and sellers. Unlike typical markets, negotiation-range mechanisms establish negotiation terms between paired bidders rather than set a final price for each transaction. This subtle difference allows single-unit heterogeneous negotiation-range markets to achieve desirable properties that cannot coexist in typical markets. This paper extends the useful properties of negotiation-range mechanisms to include coalition-resistance, making them the first markets known to offer protection from coalitions. Additionally, the notion of negotiation-range mechanisms is extended to include a restricted setting of combinatorial markets.<sup>1</sup>

## 1 Introduction

Recently much research has aimed at designing mechanisms intended to facilitate trade between market players with conflicting goals. Such mechanisms are often targeted for use on the Internet in exchanges intended to achieve efficiency by increasing sales volume and disintermediating agents. Such mechanisms must be carefully designed to motivate truthful behavior on the part of individual participants while maintaining underlying commercial viability. Though motivating individuals to behave in accordance with the market's interests is itself important; the larger goal, given the emergence of massively distributed exchanges, should also include protecting the exchange from coordinated attacks from groups of aligned individuals. This paper introduces a class of mechanisms that can be used to efficiently exchange noncommodity goods without necessarily disintermediating agents, and is the first market to look beyond the perspective of a single player by offering protection against some forms of group manipulation.

The aforementioned combinatorial market extends the concept of negotiation-based mechanisms in the context of the theory of truthful efficient markets introduced by [1]. A market consists of multiple buyers and sellers who wish to exchange goods. The market's main objective is to produce an allocation of sellers goods to buyers as to maximize the total gain from trade.

---

<sup>1</sup> This is an extended abstract, a full version is available on the author's web site.



A commonly studied model of participant behavior is taken from the field of economic mechanism design [4,12]. In this model each player has a private valuation function that assigns real values to each possible allocation. The algorithm motivates players to participate truthfully by handing payments to them.

In recent years much research has focused on designing mechanisms that allow exchange between multiple buyers and a single seller. Combinatorial auctions are a special group of such mechanisms and allow buyers to submit combinatorial requests for goods to a single seller. Almost all combinatorial auction mechanisms designed to date only apply to restricted “single minded” participants, participants who only desire a single bundle. The only polynomial non single-minded combinatorial auction mechanism known is [2].

A more general and desirable mechanism is an exchange where participants can submit both buyer bids, i.e. requests to buy items for no more than a given price, and seller bids (also called “asks”), i.e. requests to sell items for at least a given price. The mechanism in an exchange collects buyer bids and seller bids and clears the exchange by computing: (i) a set of trades, and (ii) the payments made and received by players. In designing a mechanism to compute trades and payments one must consider the bidding strategies of self-interested players, i.e. rational players that follow expected-utility maximizing strategies. Allocative efficiency is set as the primary goal; to compute a set of trades that maximizes value. In addition individual rationality (IR) is required so that all players have positive utility to participate, budget balance (BB) so that the exchange does not run at a loss, and incentive compatibility (IC) so that reporting the truth is a dominant strategy for each player.

Unfortunately, Myerson and Satterthwaite’s (1983) well known result demonstrates that in bilateral trade it is impossible to simultaneously achieve perfect efficiency, BB, and IR using an IC mechanism [11]. A unique approach to overcome Myerson and Satterthwaite’s impossibility result was attempted by [14]. This result designs both a regular and a combinatorial bilateral trade mechanism (which imposes BB and IR) that approximates truth revelation and allocation efficiency.

Negotiation-range mechanisms circumvent Myerson and Satterthwaite’s impossibility result by relaxing the requirement to set a final price. A negotiation-range mechanism does not produce payment prices for the market participants. Rather, it assigns each buyer-seller pair a price range, called a Zone Of Possible Agreements (ZOPA). The buyer is provided with the high end of the range and the seller with the low end of the range. This allows the parties to engage in negotiation over the final price while promising that the deal is beneficial for each of them.

This paper focuses on a restricted combinatorial extension to the single-unit heterogeneous setting of [1]. In that setting  $n$  sellers offer one unique good each by placing a sealed bid specifying their willingness to sell, and  $m$  buyers, interested in buying a single good each, place up to two sealed bids for goods they are interested in buying. The restricted combinatorial extension is an incentive com-

patible bilateral trade negotiation-range mechanism (ZOPAC) that is efficient, individually rational and budget balanced.

Like [1], this result does not contradict Myerson and Satterthwaite's important theorem. We present a two stage process. The first stage, ZOPAC mechanism, establishes buyer-seller matchings and negotiation terms, and the second stage allows bidders to negotiate on final prices. Although the first stage equips buyers and sellers with beneficial deals and the best negotiation terms, the second stage cannot promise that every deal will always finalize. For example if both parties in the negotiation process are aggressive negotiators they may never finalize the deal. However ZOPAC's negotiation range guarantees that each pairing is better off making a deal than not.

In a negotiation-range mechanism the range of prices each matched pair is given is resolved by a negotiation stage where a final price is determined. This negotiation stage is crucial for these mechanisms to be IC. Intuitively, a negotiation-range mechanism is incentive compatible if truth telling promises the best ZOPA from the point of view of the player in question. That is, he would tell the truth if this strategy maximizes the upper and lower bounds on his utility as expressed by the ZOPA's boundaries. Yet, when carefully examined it turns out that it is impossible (by [11]) that this goal will always hold. This is simply because such a mechanism could be easily modified to determine final prices for the players, e.g. by taking the average of the range's boundaries. Here, the negotiation stage comes into play. It is shown that if the above utility maximizing condition does not hold then the player cannot influence the negotiation bound that is assigned to his matched bidder no matter what value he declares. This means that the only thing that he may achieve by reporting a false valuation is modifying his own negotiation bound, something that he could alternatively achieve by reporting his true valuation and incorporating the effect of the modified negotiation bound into his negotiation strategy. This eliminates the benefit of reporting false valuations and allows the mechanism to compute the optimal gain from trade according to the players true values.

The problem of computing the optimal allocation that maximizes gain from trade for the heterogeneous single-unit setting can be conceptualized as the problem of finding the maximum weighted matching in a weighted bipartite graph connecting buyers and sellers, where each edge in the graph is assigned a weight equal to the difference between the respective buyer bid and seller bid. It is well known that this problem can be solved efficiently in polynomial time. However the optimal allocation which maximizes gain from trade in a combinatorial setting is NP-hard.

The combinatorial setting allows sellers of single-unit goods to join together to sell to combinatorial buyers, i.e. buyers who are interested in bundles. Buyers are not single-minded but are double-minded, i.e., can bid for two bundles. The sellers participate in a negotiation stage that resolves the ownership of the bundle to determine which seller will sell the bundle to the buyer. This setting limits the allocation problem's complexity by allowing only disjoint bundles.

A well known technique for designing IC mechanisms is the VCG payment scheme [3,8,15]. VCG IC payment schemes support efficient and IR bilateral trade but not simultaneously BB. The particular approach discussed in this paper adapts the VCG payment scheme to achieve budget balance.

In a combinatorial VCG market setting the problem of maintaining BB is compounded by the problem of maintaining IC when efficiency is approximated [10,12]. Even more so, according to [9] most “reasonable” IC combinatorial markets settings can only apply VCG and therefore are not BB. ZOPAC’s restricted setting allows efficient allocation in polynomial time, IC is maintained with the VCG approach, and BB is maintained through the modifications to VCG..

The main result is the discovery that the price ranges output by the mechanisms protect them from some group manipulations by participants. It was recently pointed out in [6] that the output prices of a mechanism and group manipulations (by coalitions) are tightly bounded. By showing that mechanisms with discriminating prices are not coalition proof and by showing that IC mechanisms that maximize efficiency require discriminating prices, i.e., IC mechanisms that maximize efficiency are not coalition proof.

Generally, coalition-proof mechanisms are difficult to achieve. A specific case of coalition resistance called group strategyproof was looked at by [5]. In a group strategyproof environment no coalition of bidders can collude by bidding untruthfully so that some members of the coalition strictly benefit without causing another member of the coalition to be strictly worse off. Such mechanisms do not resist side payments as they do not limit the extent to which a player gains from the loss of another. Negotiation-range mechanisms are coalition proof against tampering with the upper bound of the negotiation range, meaning that no coalition of buyers can improve the negotiation-starting condition of another buyer or benefit from making side payments to compensate for altering this condition.

The coalition resistance property is made possible by the negotiation range mechanism’s price ranges. To the author’s knowledge the algorithms herein are the first coalition-resistant mechanisms known for a market setting.

The rest of this paper is organized as follows. Section 2 describes the negotiation-range mechanism model and definitions. Section 3 presents the restricted combinatorial mechanism and shows that it is an IC mechanism that combines efficiency, IR and BB. Section 4 concludes with an exploration of the coalition-resistance property. The property presented applies both to the restricted combinatorial market mechanism and the single-unit heterogeneous market presented in [1].

## 2 Negotiation Markets Preliminaries

First a heterogenous market setting where buyers can buy only a single good is defined. Later this setting will be extended to a restricted combinatorial market setting. Let  $\Pi$  denote the set of players,  $N$  the set of selling players, and  $M$  the set of buying players, where  $\Pi = N \cup M$ . Let  $\Psi = \{1, \dots, k\}$  denote the set of goods.

$T_i(A_i)$  denote the  $A$ 'th coordinate of the vector  $T_i$ . Let  $T_i \in \{-1, 0, 1\}^k$  denote an *exchange vector* for a trade, such that player  $i$  buys goods  $\{A \in \Psi | T_i(A) = 1\}$  and sells goods  $\{A \in \Psi | T_i(A) = -1\}$ . Let  $T = (T_1, \dots, T_{|I|})$  denote the complete trade between all players.  $\hat{v}_j(T_j)$  defines buyer  $j$ 's *reported* value, and  $v_j(T_j)$  defines buyer  $j$ 's *true* value, for any possible exchange vector  $T_j$ .  $\hat{c}_i(T_i)$  defines seller  $i$ 's *reported* value, and  $c_i(T_i)$  defines seller  $i$ 's *true* value, for any possible exchange vector  $T_i$ . For simplicity the reported value will be assumed to be a truth-telling value and denoted  $v_j(T_j)$ ,  $c_i(T_i)$ , unless it is an assumed lie and would be denoted  $\hat{v}_j(T_j)$ ,  $\hat{c}_i(T_i)$ . All bids (buyers' and sellers') give a positive value for buying/selling a good,  $c_i(T_i) \geq 0$ , and  $v_j(T_j) \geq 0$ .  $c_i(T_z) = \infty$   $z \neq i$ , value  $\infty$  for selling anything other than her good  $A_i$ . It is assumed that buyers and sellers have no externalities, meaning that buyers and sellers are indifferent to other participants' allocations; free disposal, meaning that a buyer does not differentiate between allocations that assign him more than he requested; and normalization  $v_j(\emptyset) = 0$ . For simplicity sometimes  $v_j(T_j)$ ,  $T_j(A_i) = 1$  will be denoted as  $v_j(i)$ . Let  $T^*$  denote the value-maximizing gain from trade, given reported values,  $v_j(T_j)$ , and  $c_i(T_i)$ , from each buyer and each seller with total value  $V^* = \sum_j v_j(T_j^*) - \sum_i c_i(T_i^*)$ . Let  $\hat{T}^*$  denote the value-maximizing gain from

trade given reported lie-telling values, and  $\hat{V}^*$  the total value of  $\hat{T}^*$ . Denote by  $\chi_i$  the function that gives  $-1$  if there exists a coordinate  $A$  s.t.  $T_i(A) = -1$ , and otherwise gives  $0$ . Denote by  $\delta_i$  the function that gives  $1$  if there exists a coordinate  $A$  s.t.  $T_i(A) = 1$ , and otherwise gives  $0$ . Player  $i$  obtains at least utility  $U_i^c(c_i) = (\chi_i + 1) \cdot c_i(T_i) + L_i$  for selling and player  $j$  obtains at least utility  $U_j^v(v_j) = \delta_j \cdot v_j(T_j) - H_j$  for buying where  $L \in \mathbb{R}^N$  and  $H \in \mathbb{R}^M$  are vectors of payments and  $L_i = 0$ ,  $H_j = 0$  for  $T_i(A_i) = 0$ ,  $T_j(A_i) = 0$  respectively. Sellers are allowed to place a valuation on the good they are selling and on other goods they wish to buy, but are not allowed to bid to buy their own good.

## 2.1 Negotiation-Range Mechanisms

In designing *negotiation-range* mechanisms, the goal is to provide the player's with a range of prices within which they can negotiate the final terms of the deal by themselves. The mechanism should provide the buyer with the upper bound on the price range and the seller with the lower bound on the price range. This gives each player a promise that it will be beneficial for them to close the deal, but does not provide information about the other player's negotiation terms.

**Definition 1.** [1] *Negotiation Range: Zone Of Possible*

*Agreements, ZOPA, between a matched buyer and seller. The ZOPA is a range,  $(L, H)$ ,  $0 \leq L \leq H$ , where  $H$  is an upper bound (ceiling) price for the buyer and  $L$  is a lower bound (floor) price for the seller.*

**Definition 2.** [1] *Negotiation-Range Mechanism: A mechanism that computes a ZOPA,  $(L, H)$ , for each matched buyer and seller in  $T^*$ , and provides the buyer with the upper bound  $H$  and the seller with the lower bound  $L$ .*

One of the negotiation-range mechanism properties is *coalition proof* for buyers' ceilings.

**Definition 3.** *Ceiling Coalition Proof: A mechanism is ceiling coalition proof if the following holds for all  $S$ ,  $v$  and  $\hat{v}$  (where  $S = \{j | v_j \neq \hat{v}_j\}$  is the defecting group):  $U_j^v(v_j) \geq U_j^v(\hat{v}_j)$ ,  $\forall j \in S$ . That is, no member of the group benefits from the group's colluding and lying to the mechanism.*

### 3 The Restricted Combinatorial Setting (ZOPAC)

ZOPAC employs two distinct negotiation processes that define the behavior of individual sub-markets, a *primary market* and a *secondary market*. The primary market begins by collecting bids and determining which sellers will be allowed to sell together. The primary market then hands control over to the secondary market which uses the negotiation principle to resolve the ownership of bundles such that one seller owns each bundle. Finally, control is returned to the primary market where buyers and sellers negotiate as in ZOPAS [1] to complete their transactions.

Assume that there are  $n$  sellers, each bringing one unit of an indivisible, heterogeneous good to the market. Also assume there are  $m$  buyers, each interested in buying at most one bundle of heterogeneous goods. Each seller privately submits a bid for the good she owns and optionally declares the clique she is interested in selling with. Seller cliques are disjoint groups of sellers willing to cooperate to sell a bundle together.

**Definition 4.** *Bundle: A bundle is either a single good owned by a single seller or a collection of two or more goods owned by a clique.*

**Definition 5.** *Seller Clique: A clique  $Q$ ,  $Q \in N$  is a group of two or more sellers selling their goods as a bundle  $q$ .  $Q_i \cap Q_j = \emptyset$  for  $i \neq j$ .*

**Definition 6.** *Dominant Seller: A seller who ultimately represents a clique when negotiating with a bundle's buyer.*

For each clique, ZOPAC creates a dummy seller with a bid equal to the sum of the sellers' bids for the group's component goods. Clique  $Q_t$ 's dummy bid would be  $c_t^d(T_t^*) = \sum_{i \in Q_t} c_i(T_i)$ . Once the sellers have stated their clique membership each buyer bids on two bundles knowing that he will be assigned at most one bundle.

The mechanism now computes the ZOPAs for the primary market and establishes the negotiation terms between the buyers, sellers, and seller cliques.

The last stage is divided into three major steps. The first step computes  $T^*$  by solving the maximum weighted bipartite matching problem for the bipartite graph  $K_{q,m}$  constructed by placing the  $m$  buyers on one side of the graph, the  $q$  bundles' sellers on another and giving the edge between buyer  $j$  and bundle's

seller or clique  $i$  weight equal to  $v_j(i) - c_i^d(T_i^*)$ . The maximum weighted matching problem is solvable in polynomial time (e.g. using the Hungarian method). The result of this step is a pairing between buyers and bundles' sellers that maximizes gain from trade.

In the second step the algorithm computes a lower bound for each pairing's ZOPA (a dummy seller floor) and assigns it to the dummy seller. Each dummy seller's floor is computed by calculating the difference between the total gain from trade when the buyer is excluded and the total gain from trade of the other participants when the buyer is included (the VCG Principle).

Let  $(V_{-j})^*$  denote the gain from trade on the value maximizing trade, when player  $j$  buyer's bids are discarded but player  $j$ 's seller's or clique's bids are not discarded.

**Definition 7.** *Seller Floor: [1] The lowest price the seller should expect to receive, communicated to the seller by the mechanism. The seller floor for player  $i$  who was matched with buyer  $j$  on bundle  $A_i$ , i.e.,  $T_j(A_i) = 1$  and  $A_i \in \Psi$ , is computed as follows:  $L_i = v_j(T_j^*) + (V_{-j})^* - V^*$ . The seller is instructed not to accept less than this price from her matched buyer.*

In the third step the algorithm computes the upper bound for each pairing's ZOPA (a buyer ceiling) and assigns it to the buyer. Each buyer's ceiling is computed by removing the buyer's matched seller and calculating the difference between the total gain from trade when the buyer is excluded and the total gain from trade of the other participants when the buyer is included.

Let  $(T^{-i})^*$  denote the value-maximizing gain from trade, where seller  $i$  is removed from the trade. Denote  $(V^{-i})^*$  the total gain from trade, such that  $(V^{-i})^* = \sum_j v_j((T^{-i})_j^*) - \sum_i c_i((T^{-i})_i^*)$ . Let  $(V_{-j}^{-i})^*$  denote the gain from trade on the value maximizing trade,  $(T^{-i})^*$ , when player  $j$ 's buyer's bids are discarded but player  $j$ 's seller's or clique's bids are not discarded.

**Definition 8.** *Buyer Ceiling: [1] The highest price the buyer should expect to pay, communicated to the buyer by the mechanism. The buyer ceiling for player  $j$  on good  $A_i \in \Psi$   $i \neq j$  is computed as follows:  $H_j = v_j(T_j^*) + (V_{-j}^{-i})^* - (V^{-i})^*$  where  $T_j(A_i) = 1$ . The buyer is instructed not to pay more than this price to his matched seller.*

The third stage begins once the mechanism computes the ceiling and the floor for every bundle. The mechanism reports the floor to the seller or the clique and the ceiling to the buyer. ZOPAC now settles the deals between sellers in their cliques to determine which is the dominant seller, this process occurs in the *secondary market*. Negotiations in the secondary market are conducted by constructing a series of markets as follows:

1. Each seller submits one bid  $v_i^D$  for the entire bundle (with the exception of her own good). The seller with the highest bid becomes the current dominant seller.

2. The current dominant seller splits her bid into bids for each of the other goods in the bundle.
3. ZOPAC allocates goods to the dominant seller in a way that maximizes the gain from trade. ZOPAC also limits the dominant seller to a total final price, denoted as  $c_i^{tn}$ , that does not exceed  $v_i^D - \sum_j c_j(T_j^*)$ , where  $j$  is a seller whose good was not allocated to the dominant seller.
4. After the current dominant seller closes her deals, ZOPAC finds a new current dominant seller among the sellers still holding goods, constructs a new market representing the present state of consolidation, and the process is repeated.
5. The secondary market concludes when each clique has one seller that owns the group's goods.
6. The dominant sellers then negotiate with their bundle's original buyer to reach a final price for the bundle.

## ZOPAC

### Primary Market

For each seller  $i$

Record  $i$ 's sealed bid  $c_i(T_i^*)$  for her good  $A_i$ .

Record the seller clique  $Q_t$   $i$  is willing to sell with.

For every  $Q_t$  construct the dummy seller  $c_t^d(T_t^*) = \sum_{i \in Q_t} c_i(T_i^*)$

Each buyer submits bids for two bundles.

Compute the optimal solution  $T^*$ .

For every buyer  $j$ , allocated bundle  $t$ , compute  $L$  and  $H$  according to ZOPAS

$$L_i = v_j(t) + (V_{-j})^* - V^*$$

$$H_j = v_j(t) + (V_{-j}^{-t})^* - (V^{-t})^*$$

For every clique  $Q_t$

Reveal  $c_t^d(T_t^*)$  and  $L_t$  and find the dominant seller.

For every buyer  $j$

Report his ceiling  $H_j$  and identify his matched dominant seller  $i$ .

Conduct buyer-seller negotiation according to ZOPAS.

### Secondary Market

Construct an exchange:

1. Every seller  $i \in Q_t$  bid  $v_i^D \in [L_t - c_i(T_i^*), c_t^d(T_t^*) - c_i(T_i^*)]$

2. Choose  $i$  such that  $\max v_i^D$

3.  $i$  bids  $\tilde{v}_i(j)$  on  $j \in Q_t, j \neq i$  such that  $\sum_{j \in Q_t, j \neq i} \tilde{v}_i(j) = v_i^D$

4. Compute  $\sum_f c_f(T_f^*)$  such that  $\tilde{v}_i(f) - c_f(T_f^*) < 0$

5. Conduct negotiation between  $i$  and  $j$  such that

$$\tilde{v}_i(j) - c_j(T_j^*) \geq 0$$

$$\text{and } c_i^{tn} \leq v_i^D - \sum_f c_f(T_f^*)$$

6. Update the bound of  $i$ 's bid  $v_i^D$

Repeat 1-6 until  $\exists i$  such that  $i$  owns all goods of sellers in  $Q_t$ .

### 3.1 Analysis of the Algorithm

In this section we analyze the properties of the ZOPAC mechanism. All proofs are given in the full paper [7].

**Theorem 1.** *The ZOPAC market negotiation-range mechanism is an incentive-compatible bilateral trade mechanism that is efficient, individually rational and budget balanced.*

One can see that ZOPAC's primary market is an efficient polynomial time mechanism, efficient in the sense that players are always better off closing deals than not. The process of finding the dominant seller in a secondary market is polynomial since in every market in the secondary market at least one seller settles and is not included in the next round of consolidations. The remainder of this section shows that ZOPAC satisfies the theorem above.

*Claim.* ZOPAC is individually rational, i.e., for every dominant seller  $i$  and every buyer  $j$  in the primary market

$$U_i^c(c_t) = (\chi_t + 1) \cdot c_t^d(T_t) + L_t \geq \max \{v_i^D + c_i(T_i^*), c_t^d(T_t^*)\} \text{ and } U_j^v(v_j) = \delta_j \cdot v_j(T_j) - H_j \geq 0 \text{ and for every seller in the secondary market } U_i^c(c_i) = (\chi_i + 1) \cdot c_i(T_i) + L_i \geq c_i(T_i^*).$$

ZOPAC's budget-balanced property in the primary market follows from lemma 1 in [1] which ensures the validity of the negotiation range, i.e., that every dummy seller's floor is below her matched buyer's ceiling. In the secondary market the final price is reached between the seller's cost and the dominant seller's willingness to pay, and the mechanism only chooses valid pairings.

#### Weakly Dominant Strategies in Negotiation-Range Mechanisms.

Negotiation-range mechanisms assign bounds on prices rather than final prices and therefore a mechanism only knows each player's minimum and maximum utilities for his optimal deal. A basis for defining a weakly dominant strategy for a negotiation mechanism would then be to say that a player's minimum utility for his optimal deal should at least match the minimal utility gained with other strategies and a player's maximum utility for his optimal deal should at least match his maximum utility gained with other strategies.

The definitions for Negotiation-Range weakly dominant strategy for the primary market and the secondary market are given in the full paper [7].

The key notion of this paper is the use of negotiation as a way for buyers and sellers to determine their final price. If all conditions of the negotiation-range weakly dominant strategy always hold, in both the primary market and the secondary market, then one can set the buyer and dummy seller's final price to the average of the ceiling and the floor in the primary market and the mechanism would be incentive compatible in the traditional sense, i.e., it would not be a negotiation-range mechanism. A similar averaging can be applied in the secondary market.



The negotiation principle is required when some of the negotiation-range weakly dominant strategy conditions do not hold. In these cases it is claimed that the buyer can calculate the diverted ceiling  $\hat{H}_j$  and the seller the diverted floor  $\hat{L}_i$  (the dominant seller the diverted floor  $\hat{L}_t$  resp.). This allows a bidder to report his true value to the mechanism and use the diverted bound during the negotiation.

**Definition 9.** *A negotiation-range mechanism is incentive compatible if for every buyer  $j$  and seller  $i$  in the mechanism, (primary market or secondary market) truth-telling is their negotiation-range weakly dominant strategy or buyer  $j$  (resp. seller  $i$ ) can compute  $\hat{H}_j$  (resp.  $\hat{L}_i$ , or  $\hat{L}_t$ ) and use the computed value during the negotiation.*

**Theorem 2.** *ZOPAC is an incentive compatible negotiation-range mechanism for buyers and dominant sellers in the primary market, and for sellers in their secondary market.*

The proof of theorem 2 is compounded by number of claims, each proving a minimum or maximum utility condition for the cases where the allocation changed or not as a result of the player lie.

## 4 Coalition-Resistent Market

This section shows another advantage of leaving the matched buyers and sellers with a negotiation range rather than a final price. The negotiation range also “protects” the mechanism from some manipulations. For simplicity this section will refer to a heterogeneous setting. The findings apply to ZOPAC’s setting and to ZOPAS [1]. All the proofs of this section are given in the full paper [7].

In a market there are numerous ways participants can collude to improve their gain. Buyers can collude, sellers can collude or a mixed coalition of buyers and sellers can work together. This section deals with coalitions of buyers.

In a negotiation-range mechanism there are two values that a coalition can attempt to divert; the negotiation ceiling and floor. Since the floors are essentially VCG it is impossible, and known, that they cannot resist manipulation, i.e., it is possible for a losing buyer to impact the floor of the seller matched with the winning buyer. However, it is impossible for any buyer to help another buyer by manipulating his ceiling.

In section 3 it was shown that whenever the negotiation-range weakly dominant strategy does not hold, buyer  $j$  can calculate his diverted ceiling given the ceiling resulting from his truth-telling strategy. The basis of the coalition-proof property is that no other lie stated by the coalition members can lead to a lower ceiling than  $j$ ’s calculated ceiling. In other words  $j$  can help himself more than any other buyer in the market, so no coalition of buyers can collude by bidding untruthfully so that some members of the coalition can improve their calculated negotiation terms, i.e., their ceiling. The following definitions formalize the coalition-resistent property in negotiation-range markets.

Let  $\bar{H}_z$  be the ceiling calculated by buyer  $z$  for the negotiation phase when the mechanism assigned the ceiling  $H_z$  to  $z$ . Let  $\hat{H}_z^{(j)}$  be the ceiling given by the mechanism when buyer  $j$  lies in order to help buyer  $z$ .

**Definition 10.** A negotiation-range market is ceiling coalition proof for buyers if: No coalition of buyers can collude by bidding untruthfully,  $S = \{j | v_j(T_j^*) \neq \hat{v}_j(T_j^*)\}$ , so that some member of the coalition  $z \in S$  improves his calculated ceiling utility, i.e.,  $\forall z, T_z^*(A_g) = 1, \hat{T}_z^*(A_f) = 1, j \in S, v_z(g) - \bar{H}_z \geq v_z(f) - \hat{H}_z^{(j)}$ .

**Theorem 3.** ZOPAC and ZOPAS [1] are negotiation-range markets that are ceiling coalition proof for buyers. That is, for every buyer  $j$  and every buyer  $z$  such that  $j \neq z, T_j^*(A_i) = 1, T_z^*(A_g) = 1, \hat{T}_z^*(A_f) = 1$  and for every lie buyer  $j$  can tell,  $v_j(i) \neq \hat{v}_j(i)$  or  $v_j(k) \neq \hat{v}_j(k)$ ,  $z$  cannot improve his calculated ceiling utility.

The proof of theorem 3 begins by showing that buyer  $j$  cannot help buyer  $z$  if the optimal allocation is not affected by  $j$ 's lie.

*Claim.* Let  $j$  be a buyer matched to seller  $i$  and let  $z$  be a buyer matched to seller  $g$  in  $T^*$  and in  $\hat{T}^*$ . Then,

$$\bar{H}_z \leq \hat{H}_z^{(j)}. \quad (1)$$

The proof of the claim begins by showing that the amount by which buyer  $z$  can reduce his ceiling is more than the amount buyer  $j$  can change his own valuation when lying. The proof of the claim continues by showing that all possible allocations that pair buyer  $j$  with seller  $i$  imply (1).

Let  $\bar{H}_z = H_z - v_z(g) + \bar{v}_z(g) = \bar{v}_z(g) + (V_{-z}^{-g})^* - (V^{-g})^*$  where  $\bar{v}_z(g)$  is the value that  $z$  uses to compute his ceiling. Recall that  $\hat{H}_z^{(j)} = v_z(g) + (\hat{V}_{-z}^{-g})^* - (\hat{V}^{-g})^*$  where  $(\hat{T}_{-z}^{-g})^*$  and  $(\hat{T}^{-g})^*$  are the allocations resulting from  $j$ 's lie. It follows that in order to prove (1) it is necessary to show

$$(V_{-z}^{-g})^* + (\hat{V}^{-g})^* \leq v_z(g) - \bar{v}_z(g) + (\hat{V}_{-z}^{-g})^* + (V^{-g})^*. \quad (2)$$

The proof of (2) is as follows. Given the allocations  $(\hat{T}^{-g})^*$  and  $(T_{-z}^{-g})^*$ , which together have value equal to  $(\hat{V}^{-g})^* + (V_{-z}^{-g})^*$ , one can use the same pairs matched in these allocations to create a new pair of valid allocations. Consequently the sum of values of the new allocations and the sum of values of the original pair of allocations is the same. It is also required that one of the new allocations does not include seller  $g$  and is based on the true valuation, while the other allocation does not include buyer  $z$  nor seller  $g$  and is based on false valuation. This means that the sum of values of these new allocations is at most  $(V^{-g})^* + (\hat{V}_{-z}^{-g})^*$ , which proves (2).

The proof of theorem 3's is concluded by showing that buyer  $j$  cannot help buyer  $z$  even when the optimal allocation changes as a result of  $j$ 's lie.

*Claim.* Let  $j$  be a buyer matched to seller  $i$  and let  $z$  be a buyer matched to seller  $g$  in  $T^*$ , and let  $k \neq i$  and  $f$  be sellers matched with  $j$  and  $z$  respectively in  $\hat{T}^*$ . Then,

$$v_z(g) - \bar{H}_z \geq v_z(f) - \hat{H}_z^{(j)}. \quad (3)$$

## 5 Conclusions and Extensions

This paper showed that negotiation-range mechanisms exhibit coalition resistance, and introduced a restricted combinatorial setting.

The most general and desirable combinatorial market would allow each seller to sell a bundle of goods and allow buyers to buy bundles that are not necessarily the exact bundles for sale. This mechanism would require a complex negotiation scheme between the sellers.

Coalition proof is an interesting property of the ZOPAS and ZOPAC bilateral trade mechanisms, but in the present embodiment the floors are not coalition proof in the same way that VCG mechanisms are not coalition proof. An interesting direction for future research is the question: Can one define the floor on a basis other than VCG such that the new floor would be coalition proof as well?

**Acknowledgements.** The author would like to gratefully acknowledge Colin Brady for his helpful simulation application, discussions, comments, and suggestions.

## References

1. Y. Bartal, R. Gonen, and P. La Mura. Negotiation-Range Mechanisms: Exploring the Limits of Truthful Efficient Markets. *Proc. of 5th EC*, pp. 1-8, 2004.
2. Y. Bartal, R. Gonen, and N. Nisan. Incentive Compatible Multi-Unit Combinatorial Auctions. *Proc. of 9th TARK*, pp. 72-87, 2003.
3. E. H. Clarke Multipart Pricing of Public Goods. *Journal of Public Choice* 1971, volume 2, pp. 17-33.
4. J. Feigenbaum, C. Papadimitriou, and S. Shenker. Sharing the Cost of Multicast Transmissions. *Journal of Computer and System Sciences*, 63(1), 2001.
5. A.V. Goldberg, and J.D. Hartline. Envy-Free Auctions for Digital Goods. *Proc. of the 4th EC*, pp. 29-35, 2003.
6. R. Gonen. Coalition Resistance. *Preliminary version* 2003.
7. R. Gonen. Negotiation-Range Mechanisms: Coalition Resistance Markets. <http://www.cs.huji.ac.il/rgonen>.
8. T. Groves Incentives in teams. *Journal of Econometrica* 1973, volume 41, pp. 617-631.
9. R. Lavi, A. Mu'alem and N. Nisan. Towards a Characterization of Truthful Combinatorial Auctions. *Proc. of 44th FOCS*, 2003.
10. D. Lehmann, L. I. O'Callaghan, and Y. Shoham. Truth revelation in approximately efficient combinatorial auctions. *Journal of the ACM*, 49(5), pp. 577-602, 2002.
11. R. Myerson, M. Satterthwaite. Efficient Mechanisms for Bilateral Trading. *Journal of Economic Theory*, 28, pp. 265-81, 1983.
12. N. Nisan and A. Ronen. Algorithmic Mechanism Design. In *Proc. of STOC, 1999*
13. C. Papadimitriou Algorithm, Games, and the Internet. *Proc. of STOC 2001*.
14. D.C. Parkes, J. Kalagnanam, and M. Eso. Achieving Budget-Balance with Vickrey-Based Payment Schemes in Exchanges. *Proc. of 17th IJCAI*, pp. 1161-1168, 2001.
15. W. Vickrey Counterspeculation, Auctions and Competitive Sealed Tenders. In *Journal of Finance* 1961, volume 16, pp. 8-37.

# Approximation Algorithms for Quickest Spanning Tree Problems

Refael Hassin<sup>1</sup> and Asaf Levin<sup>2</sup>

<sup>1</sup> Department of Statistics and Operations Research, Tel-Aviv University, Tel-Aviv 69978, Israel. hassin@post.tau.ac.il

<sup>2</sup> Faculty of Industrial Engineering and Management, The Technion, Haifa 32000, Israel. levinas@tx.technion.ac.il

**Abstract.** Let  $G = (V, E)$  be an undirected multi-graph with a special vertex  $root \in V$ , and where each edge  $e \in E$  is endowed with a length  $l(e) \geq 0$  and a capacity  $c(e) > 0$ . For a path  $P$  that connects  $u$  and  $v$ , the *transmission time* of  $P$  is defined as  $t(P) = \sum_{e \in P} l(e) + \max_{e \in P} \frac{1}{c(e)}$ . For a spanning tree  $T$ , let  $P_{u,v}^T$  be the unique  $u - v$  path in  $T$ . The QUICKEST RADIUS SPANNING TREE PROBLEM is to find a spanning tree  $T$  of  $G$  such that  $\max_{v \in V} t(P_{root,v}^T)$  is minimized. In this paper we present a 2-approximation algorithm for this problem, and show that unless  $P = NP$ , there is no approximation algorithm with performance guarantee of  $2 - \epsilon$  for any  $\epsilon > 0$ . The QUICKEST DIAMETER SPANNING TREE PROBLEM is to find a spanning tree  $T$  of  $G$  such that  $\max_{u,v \in V} t(P_{u,v}^T)$  is minimized. We present a  $\frac{3}{2}$ -approximation to this problem, and prove that unless  $P = NP$  there is no approximation algorithm with performance guarantee of  $\frac{3}{2} - \epsilon$  for any  $\epsilon > 0$ .

## 1 Introduction

Let  $G = (V, E)$  be an undirected multi-graph where each edge  $e \in E$  is endowed with a length  $l(e) \geq 0$  and a capacity  $c(e) > 0$ . For an edge  $e = (u, v) \in E$ , the *capacity* of  $e$  is the amount of data that can be sent from  $u$  to  $v$  along  $e$  in a single time unit. The *reciprocal capacity* along  $e$  is  $r(e) = \frac{1}{c(e)}$ . The *length* of  $e$  is the time required to send data from  $u$  to  $v$  along  $e$ . If we want to transmit a data of size  $\sigma$  along  $e$ , then the time it takes is the *transmission time*,  $t(e) = l(e) + \frac{\sigma}{c(e)}$ . For a path  $P$  that connects  $u$  and  $v$  the *length* of  $P$  is defined as  $l(P) = \sum_{e \in P} l(e)$ , the *reciprocal capacity* of  $P$  is defined as  $r(P) = \max_{e \in P} r(e)$ , and the *transmission time* of  $P$  is defined as  $t(P) = l(P) + \sigma r(P)$ . Denote  $n = |V|$  and  $m = |E|$ .

A path  $P$  that connects  $u$  and  $v$  is a  $u - v$  *path*. For a spanning tree  $T$ , let  $P_{u,v}^T$  be the unique  $u - v$  path in  $T$ .

The QUICKEST PATH PROBLEM (QPP) is to find for a given pair of vertices  $u, v \in V$  a  $u - v$  path  $P$ , whose transmission time is minimized. Moore [8] introduced this problem and presented an  $O(m^2 + mn \log n)$ -time algorithm that solves the problem for all possible values of  $\sigma$ . Chen and Chin [4] showed an  $O(m^2 + mn \log n)$ -time algorithm that solves the problem for a common source

$u$  and all destinations  $v$  for all possible values of  $\sigma$ . Their algorithm creates a data structure that encodes the quickest paths for all possible values of  $\sigma$ . Using this data-structure, it is possible to obtain an optimal path for a single value of  $\sigma$  in  $O(\log n)$  time (after the  $O(m^2 + mn \log n)$ -time preprocessing). Martins and Santos [7] provided a different algorithm for the same problem with the same time-complexity and a reduced space complexity. Rao [10] considered the problem with a single value of  $\sigma$  and a given pair  $u, v$ . He designed a probabilistic algorithm with running time  $O(pm + pn \log n)$  that has a high probability (that depends on  $p$ ) of computing a path with transmission time close to the optimal one. The all-pairs shortest paths variant of this problem is solved in  $O(mn^2)$  time (after the  $O(m^2 + mn \log n)$ -time preprocessing) for all possible values of  $\sigma$  and moreover, the algorithm constructs a data structure such that it is possible to get an optimal path for a single value of  $u, v$  and  $\sigma$  in  $O(\log n)$  time. This algorithm to solve the all-pairs variant appeared in both Chen and Hung [1] and Lee and Papadopolou [6]. Rosen, Sun and Xue [11] and Pelegrin and Fernandez [9] considered the bi-criteria minimization problem of finding a  $u - v$  path whose length and reciprocal capacity are both minimized. These objectives are sometimes contradicting, and [11,9] presented algorithms that compute the efficient path set. Chen [2,3] and Rosen, Sun and Xue [11] considered the  $k$  quickest simple path problem.

In this paper we consider problems with a single value of  $\sigma$ . W.l.o.g. we can assume that  $\sigma = 1$  (otherwise we can scale  $c$ ). Therefore, for a  $u - v$  path  $P$ ,  $t(P) = l(P) + r(P)$ .

Many network design problems restrict the solution network to be a tree (i.e., a subgraph without cycles). Usually such a constraint results from the communication protocols used in the network. In broadcast networks, one usually asks for a small maximum delay of a message sent by a *root* vertex to all the users (all the other vertices) of the network. In other communication networks, one seeks a small upper bound on the delay of transmitting a message between any pair of users (vertices). These objectives lead to the following two problems:

The **QUICKEST RADIUS SPANNING TREE PROBLEM** is to find a spanning tree  $T$  of  $G$  such that  $rad_t(T) = \max_{v \in V} t(P_{root,v}^T)$  is minimized.

The **QUICKEST DIAMETER SPANNING TREE PROBLEM** is to find a spanning tree  $T$  of  $G$  such that  $diam_t(T) = \max_{u,v \in V} t(P_{u,v}^T)$  is minimized.

The *shortest paths tree* (SPT) is defined as follows: given an undirected multi-graph  $G = (V, E)$  with a special vertex  $root \in V$ , and where each edge  $e \in E$  is endowed with a length  $l(e) \geq 0$ , the SPT is a spanning tree  $T = (V, E_T)$  such that for every  $u \in V$  the unique  $root - u$  path in  $T$  has the same total length as the length of the shortest path in  $G$  that connects  $root$  and  $u$ . This tree can be found in polynomial time by applying Dijkstra's algorithm from  $root$ . We denote the returned tree by  $SPT(G, root, l)$ .

To define the absolute 1-center of a graph  $G = (V, E)$  where each edge  $e \in E$  is endowed with a length  $l(e) \geq 0$ , we refer also to interior points on an edge by their distances (along the edge) from the two end-vertices of the edge. We let  $A(G)$  denote the continuum set of points on the edges of  $G$ .  $l$  induces a length

function over  $A(G)$ . For  $x, y \in A(G)$ , let  $l_G(x, y)$  denote the length of a shortest path in  $A(G)$  connecting  $x$  and  $y$ . The *absolute 1-center* is a minimizer  $x$  of the function  $F(x) = \max_{v \in V} l_G(x, v)$ .

The MINIMUM DIAMETER SPANNING TREE PROBLEM is defined as follows: given an undirected multi-graph  $G = (V, E)$  where each edge is endowed with a length  $l(e) \geq 0$ , we want to find a spanning tree  $T = (V, E_T)$  such that  $\text{diam}_l(T) = \max_{u, v \in V} l(P_{u, v}^T)$  is minimized. This problem is solved [5] in polynomial time by computing a shortest path tree from an absolute 1-center of  $G$ .

### Our results:

1. A 2-approximation algorithm for the QUICKEST RADIUS SPANNING TREE PROBLEM.
2. For any  $\epsilon > 0$ , unless  $P = NP$  there is no approximation algorithm with performance guarantee of  $2 - \epsilon$  for the QUICKEST RADIUS SPANNING TREE PROBLEM.
3. A  $\frac{3}{2}$ -approximation algorithm for the QUICKEST DIAMETER SPANNING TREE PROBLEM.
4. For any  $\epsilon > 0$ , unless  $P = NP$  there is no approximation algorithm with performance guarantee of  $\frac{3}{2} - \epsilon$  for the QUICKEST DIAMETER SPANNING TREE PROBLEM.

## 2 A 2-Approximation Algorithm for the Quickest Radius Spanning Tree Problem

Algorithm *quick\_radius*:

1. For every  $u \in V \setminus \{\text{root}\}$ , compute  $QP_{\text{root}, u}$ , that is a quickest path from *root* to  $u$  in  $G$ .
2. Let  $G' = (V, E') = \bigcup_{u \in V \setminus \{\text{root}\}} QP_{\text{root}, u}$ .
3.  $T' = \text{SPT}(G', \text{root}, l)$ .
4. **Return**  $T'$ .

*Example 1.* Consider the graph  $G = (\{\text{root}, 1, 2, 3\}, E_1 \cup E_2)$ , where  $E_1 = \{(\text{root}, 1), (1, 2)\}$  and for each  $e \in E_1$   $l(e) = 1$ ,  $r(e) = 2$ , and  $E_2 = \{(\text{root}, 2), (2, 3)\}$  and for each  $e \in E_2$   $l(e) = 0$  and  $r(e) = 3\frac{1}{2}$ . Then,  $QP_{\text{root}, 1} = (\text{root}, 1)$ ,  $QP_{\text{root}, 2} = (\text{root}, 1, 2)$  and  $QP_{\text{root}, 3} = (\text{root}, 2, 3)$ . Therefore,  $G'$  that is computed in Step 2 of Algorithm *quick\_radius* is exactly  $G$ . In Step 3 we compute the tree  $T' = (\{\text{root}, 1, 2, 3\}, E_{T'})$  where  $E_{T'} = \{(\text{root}, 1), (\text{root}, 2), (2, 3)\}$ , and therefore the cost of  $T'$  is defined by the path  $P = (\text{root}, 2, 3)$  where  $r(P) = r(\text{root}, 2) = 3\frac{1}{2}$ ,  $l(P) = 1 + 0 = 1$ ,  $t(P) = r(P) + l(P) = 3\frac{1}{2} + 1 = 4\frac{1}{2}$ . Note that  $T'$  is the optimal solution.

The time complexity of Algorithm *quick\_radius* is  $O(m^2 + mn \log m)$ . It is dominated by the time complexity of Step 1 that is  $O(m^2 + mn \log m)$  by Chen and Chin [4]. The algorithm returns a spanning tree of  $G$ , and therefore it is feasible.

Before we prove the performance guarantee of the algorithm we remark that as in Example 1  $G'$  may contain cycles. The reason for this is that the property of shortest paths, “any sub-path of a shortest path must itself be a shortest path” does not hold for quickest paths (as noted in [4]).

**Theorem 1.** *Algorithm `quick_radius` is a 2-approximation algorithm for the QUICKEST RADIUS SPANNING TREE PROBLEM.*

### 3 Inapproximability of the Quickest Radius Spanning Tree Problem

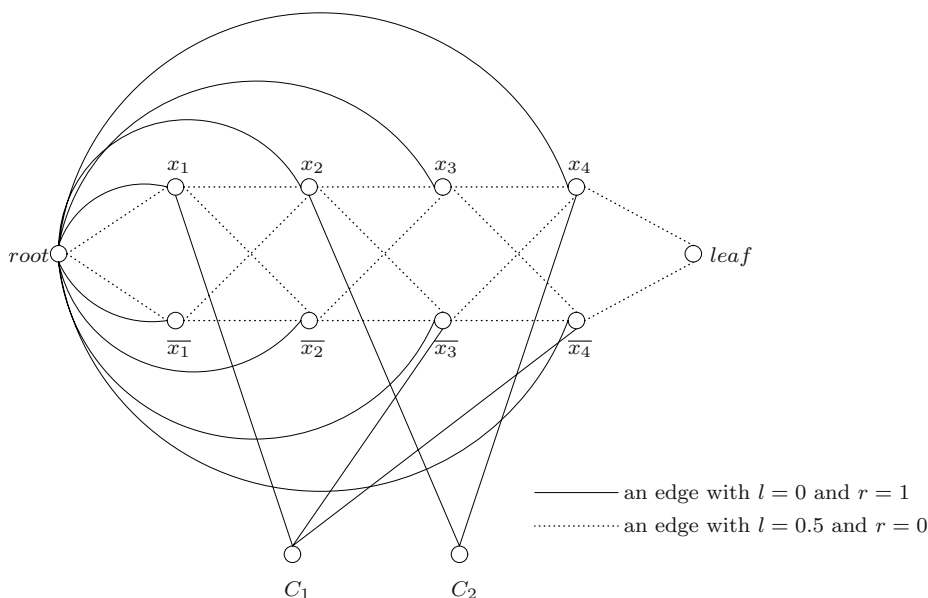
In this section we show that unless  $P = NP$ , there is no approximation algorithm with a performance guarantee of  $2 - \epsilon$  for any  $\epsilon > 0$ . In order to motivate our final reduction, we first present a simpler reduction that shows that there is no approximation algorithm with a performance guarantee of  $\frac{3}{2} - \epsilon$ . Next, we show an example where the fact that Algorithm `quick_radius` chooses its output from the edge set of the quickest paths, does not allow it to have a performance guarantee of better than 2. Finally, we present the main result of this section by combining the ideas of its first two parts.

#### 3.1 A $\frac{3}{2}$ Lower Bound on the Approximation Ratio

**Lemma 1.** *If  $P \neq NP$ , then there is no  $(\frac{3}{2} - \epsilon)$ -approximation algorithm for the QUICKEST RADIUS SPANNING TREE PROBLEM for any  $\epsilon > 0$ .*

*Proof.* We prove the claim via reduction from SAT. Let  $C_1, C_2, \dots, C_m$  be the clauses of a SAT instance in the variables  $x_1, x_2, \dots, x_n$ . We construct a graph  $G = (V, E)$  as follows (see Figure 1):  $V = \{root, leaf\} \cup \{x_i, \bar{x}_i : 1 \leq i \leq n\} \cup \{C_j : C_j \text{ is a clause}\}$ , i.e., we have special pair of vertices  $root$  and  $leaf$ , we have a vertex for each variable and another for its negation, and a vertex for each clause.  $E = E_1 \cup E_2 \cup E_3$ .  $E_1 = \{(root, x_1), (root, \bar{x}_1)\} \cup \{(x_i, x_{i+1}), (x_i, \bar{x}_{i+1}), (\bar{x}_i, x_{i+1}), (\bar{x}_i, \bar{x}_{i+1}) : 1 \leq i \leq n-1\} \cup \{(x_n, leaf), (\bar{x}_n, leaf)\}$ , where for every  $e \in E_1$ ,  $l(e) = 0$  and  $r(e) = 1$ .  $E_2 = \{(l_i, C_j) : \text{for every literal } l_i \text{ of } C_j \forall j\}$ ,  $E_3 = \{(root, x_i), (root, \bar{x}_i) : 1 \leq i \leq n\}$ , where for every  $e \in E_2 \cup E_3$ ,  $l(e) = 0.5$  and  $r(e) = 0$ . This instance contains parallel edges, however it is easy to make it a graph by splitting edges. This defines the instance for the QUICKEST RADIUS SPANNING TREE PROBLEM.

Assume that the formula is satisfied by some truth assignment. We now show how to construct a spanning tree  $T = (V, E_T)$  such that  $rad_t(T) = 1$ .  $E_T = E_T^1 \cup E_T^2$ .  $E_T^1$  is a path of edges from  $E_1$  that connects  $root$  and  $leaf$  where all its intermediate vertices are false literals in the truth assignment.  $E_T^2$  is composed of the edges  $(root, l_i)$  from  $E_3$  for all the true literals  $l_i$ , also each clause contains at least one true literal, and we add to  $E_T^2$  an edge between the clause vertex and such a true literal (exactly one edge for each clause vertex). Then,  $rad_t(T) = 1$  because for every  $u \in V$  that is on the path of  $E_T^1$  edges, we



**Fig. 1.** Example for the construction of Lemma 1:  $C_1 = x_1 \vee \bar{x}_3 \vee \bar{x}_4$  and  $C_2 = x_2 \vee x_4$

have a path whose transmission time is exactly 1, and for every other vertex we have a path from *root* of length at most 1 and reciprocal capacity 0, so that its transmission time is at most 1.

We now show that if the formula cannot be satisfied, then for every tree  $T = (V, E_T)$ ,  $rad_t(T) \geq \frac{3}{2}$ . If  $P_{root, leaf}^T$  contains an edge from  $E_3$  (at least one), then its length is at least  $\frac{1}{2}$ , its reciprocal capacity is 1, and therefore, its transmission time is at least  $\frac{3}{2}$ . Assume now that  $P_{root, leaf}^T$  is composed only of edges from  $E_1$ . Since the formula cannot be satisfied, then for every such path  $P_{root, leaf}^T$  there is a clause vertex  $C_j$  all of whose literals are in  $P_{root, leaf}^T$ . For this  $C_j$ ,  $P_{root, C_j}^T$  has length at least  $\frac{1}{2}$ , and its reciprocal capacity is 1. Therefore, its transmission time is at least  $\frac{3}{2}$ .

For any  $\epsilon > 0$ , a  $(\frac{3}{2} - \epsilon)$ -approximation algorithm distinguishes between 1 and  $\frac{3}{2}$ , and therefore solves the SAT instance.  $\square$

### 3.2 On Using Only Quickest Path Edges

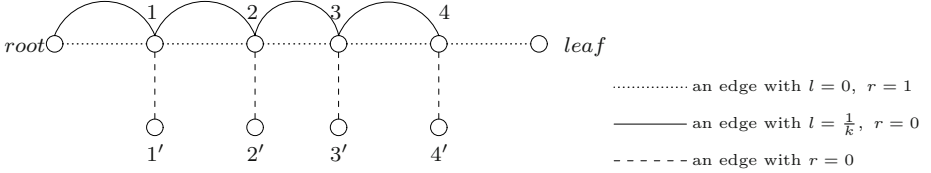
Recall that  $G'$  is the union of a quickest  $root - u$  path in  $G$  for all  $u \in V \setminus \{root\}$ . Let  $\epsilon > 0$  and denote by  $OPT$  the cost of an optimal solution. In this subsection we show an instance where any spanning tree  $T'$  of  $G'$  satisfies  $rad_t(T') \geq (2 - \epsilon)OPT$ .

Let  $\delta > 0$  be small enough, and let  $k$  be a large enough integer. We construct a graph  $G = (V, E)$  as follows:  $V = \{root, leaf\} \cup \{i, i' : 1 \leq i \leq k - 1\}$ .



$E = E_1 \cup E_2 \cup E_3 \cup E_4$ .  $E_1 = \{(root, 1)\} \cup \{(i, i+1) : 1 \leq i \leq k-2\} \cup \{(k-1, leaf)\}$  where for each edge  $e \in E_1$ ,  $l(e) = 0$  and  $r(e) = 1$ .  $E_2 = \{(root, 1)\} \cup \{(i, i+1) : 1 \leq i \leq k-2\}$  where for each edge  $e \in E_2$ ,  $l(e) = \frac{1}{k}$  and  $r(e) = 0$ .  $E_3 = \{(i, i') : 1 \leq i \leq k-1\}$ , where for each  $e_i = (i, i')$ ,  $l(e_i) = \frac{k-i}{k}$  and  $r(e_i) = 0$ .  $E_4 = \{(root, i') : 1 \leq i \leq k-1\}$  where for each edge  $e \in E_4$ ,  $l(e) = 1 + \delta$  and  $r(e) = 0$ .

The quickest paths from  $root$  use only the edges from  $E_1 \cup E_2 \cup E_3$ , and the subgraph  $G'$  for  $k = 5$  is shown in Figure 2.



**Fig. 2.** The graph  $G'$  for the example in Subsection 3.2

We note that in this graph the tree  $T = (V, E_T)$  where  $E_T = E_1 \cup E_4$ , has  $rad_t(T) = 1 + \delta$ . Consider a spanning tree  $T' = (V, E_{T'})$  where  $E_{T'} \subseteq E_1 \cup E_2 \cup E_3$ . Then, if  $P_{root, leaf}^{T'}$  is composed of all the edges of  $E_2$  and the edge  $(k-1, leaf) \in E_1$ , then  $rad_t(T') \geq 2 - \frac{1}{k}$ . Otherwise,  $P_{root, leaf}^{T'}$  contains other edges from  $E_1$ . Denote  $0 = root$ , and let  $(i-1, i) \in P_{root, leaf}^{T'}$  be an edge from  $E_1$  such that  $i$  is minimized ( $i \geq 1$ ). Then,  $P_{root, i'}^{T'}$  is composed of the edge  $(i, i')$  and the part of  $P_{root, leaf}^{T'}$  that connects  $root$  and  $i$ . Therefore, its length is at least  $1 - \frac{1}{k}$  (by the minimality of  $i$ ), and its reciprocal capacity is 1. Therefore,  $rad_t(T') \geq 2 - \frac{1}{k}$ .

We pick  $\delta$  and  $k$  such that  $\frac{2-\frac{1}{k}}{1+\delta} \geq 2 - \epsilon$ .

This example also shows that our analysis of Algorithm *quick\_radius* is tight.

### 3.3 A Tight Lower Bound on the Approximability Ratio

In this subsection we show that unless  $P = NP$  there is no polynomial time approximation algorithm that guarantees a ratio of  $2 - \epsilon$  for any  $\epsilon > 0$ . Since Algorithm *quick\_radius* is a 2-approximation algorithm, it is the **best possible**.

**Theorem 2.** *If  $P \neq NP$ , then there is no  $(2 - \epsilon)$ -approximation algorithm for the QUICKEST RADIUS SPANNING TREE PROBLEM for any  $\epsilon > 0$ .*

## 4 The Quickest Diameter Spanning Tree Problem

In this section we consider the QUICKEST DIAMETER SPANNING TREE PROBLEM. We present a  $\frac{3}{2}$ -approximation algorithm, and prove that unless  $P = NP$  this is the **best possible**.

Denote by  $OPT$  the cost of an optimal solution,  $T_{opt} = (V, E_{opt})$ , to the QUICKEST DIAMETER SPANNING TREE PROBLEM. Denote by  $MDT(G, l)$  the minimum diameter spanning tree of a graph  $G$  where the edges are endowed with a length function  $l$ .

Algorithm *quick\_diameter*:

1. For every  $r \in \{r(e) : e \in E\}$  do
  - a) Let  $G_r = (V, E_r)$ , where  $E_r = \{e \in E : r(e) \leq r\}$ .
  - b)  $T'_r = MDT(G_r, l)$ .
2. Let  $T'$  be a minimum cost solution among  $\{T'_r\}_{r \in \{r(e) : e \in E\}}$ .
3. **Return**  $T'$ .

**Lemma 2.** *Let  $r_{max} = \max_{e \in E_{opt}} r(e)$ . For every pair of vertices  $u, v \in V$ ,  $\frac{l(P_{u,v}^{T_{opt}})}{2} + r_{max} \leq OPT$ .*

*Proof.* The claim clearly holds if the path  $P_{u,v}^{T_{opt}}$  contains an edge  $e$  with  $r(e) = r_{max}$ . Assume otherwise (that the claim does not hold), and let  $y \in P_{u,v}^{T_{opt}}$  be such that there exists  $z \notin P_{u,v}^{T_{opt}}$ ,  $P_{y,z}^{T_{opt}}$  contains an edge  $e$  with  $r(e) = r_{max}$ , and  $P_{u,v}^{T_{opt}}$  and  $P_{y,z}^{T_{opt}}$  are edge-disjoint. Then,  $2OPT \geq t(P_{u,z}^{T_{opt}}) + t(P_{v,z}^{T_{opt}}) \geq l(P_{u,y}^{T_{opt}}) + l(P_{v,y}^{T_{opt}}) + 2r_{max} = l(P_{u,v}^{T_{opt}}) + 2r_{max}$ , and the claim follows.  $\square$

For a tree  $T$ , denote by  $diam_l(T) = \max_{u,v \in V} l(P_{u,v}^T)$ .

**Theorem 3.** *Algorithm *quick\_diameter* is a  $\frac{3}{2}$ -approximation algorithm for the QUICKEST DIAMETER SPANNING TREE PROBLEM.*

*Proof.* Algorithm *quick\_diameter* is polynomial, and returns a feasible solution. It remains to show the approximation ratio of the algorithm.

1. By the optimality of  $T'_{r_{max}}$  (for the minimum diameter spanning tree problem),  $diam_l(T'_{r_{max}}) \leq diam_l(T_{opt})$ .
2. By Lemma 2,  $diam_l(T_{opt}) \leq 2OPT - 2r_{max}$ . Therefore, by 1,  $diam_l(T'_{r_{max}}) \leq 2OPT - 2r_{max}$ .
3. Since  $T'_{r_{max}} \in G_{r_{max}}$ ,  $r(T'_{r_{max}}) \leq r_{max}$ .
4. By 2 and 3,  $diam_t(T'_{r_{max}}) = diam_l(T'_{r_{max}}) + r(T'_{r_{max}}) \leq 2OPT - r_{max}$ .
5. By 1,  $diam_l(T'_{r_{max}}) \leq diam_l(T_{opt}) \leq OPT$ .
6. By 5,  $diam_t(T'_{r_{max}}) \leq OPT + r_{max}$ .
7. By 4 and 6  $diam_t(T'_{r_{max}}) \leq \frac{3}{2}OPT$ . Therefore,  $diam_t(T') \leq \frac{3}{2}OPT$ .

$\square$

**Theorem 4.** *If  $P \neq NP$ , then there is no  $(\frac{3}{2} - \epsilon)$ -approximation algorithm for the QUICKEST DIAMETER SPANNING TREE PROBLEM for any  $\epsilon > 0$ .*

## 5 Discussion

In this paper we consider a pair of new problems that consider the transmission time along paths instead of the usual length of the paths. The two problems that we consider are to minimize the diameter and the radius of a spanning tree with respect to the transmission time. There are numerous other graph problems of interest that ask to compute a minimum cost subgraph where the cost is a function of the distances among vertices on the subgraph. Defining these problems under the transmission time definition of length opens an interesting area for future research.

## References

1. G.-H. Chen and Y.-C. Hung, "On the quickest path problem," *Information Processing Letters*, **46**, 125-128, 1993.
2. Y. L. Chen, "An algorithm for finding the  $k$  quickest paths in a network," *Computers & Opns. Res.*, **20**, 59-65, 1993.
3. Y. L. Chen, "Finding the  $k$  quickest simplest paths in a network," *Information Processing Letters*, **50**, 89-92, 1994.
4. Y. L. Chen and Y. H. Chin, "The quickest path problem," *Computers & Opns. Res.*, **17**, 153-161, 1990.
5. R. Hassin and A. Tamir, "On the minimum diameter spanning tree problem," *Information Processing Letters*, **53**, 109-111, 1995.
6. D. T. Lee and E. Papadopoulou, "The all-pairs quickest path problem," *Information Processing Letters*, **45**, 261-267, 1993.
7. E. Q. V. Martins and J. L. E. Santos, "An algorithm for the quickest path problem," *Operations Research Letters*, **20**, 195-198, 1997.
8. M. H. Moore, "On the fastest route to convoy-type traffic in flowrate-constrained networks," *Transportation Science*, **10**, 113-124, 1976.
9. B. Pelegriin and P. Fernandez, "On the sum-max bicriterion path problem," *Computers & Opns. Res.*, **25**, 1043-1054, 1998.
10. N. S. V. Rao, "Probabilistic quickest path algorithm," *Theoretical Computer Science*, **312**, 189-201, 2004.
11. J. B. Rosen, S. Z. Sun and G. L. Xue, "Algorithms for the quickest path problem and enumeration of quickest paths," *Computers & Opns. Res.*, **18**, 579-584, 1991.

# An Approximation Algorithm for Maximum Triangle Packing

R. Hassin and S. Rubinstein

Department of Statistics and Operations Research, Tel Aviv University,  
Tel Aviv, 69978, Israel.

{hassin,shlomiru}@post.tau.ac.il

**Abstract.** We present a randomized  $(\frac{89}{169} - \epsilon)$ -approximation algorithm for the weighted maximum triangle packing problem, for any given  $\epsilon > 0$ . This is the first algorithm for this problem whose performance guarantee is better than  $\frac{1}{2}$ . The algorithm also improves the best known approximation bound for the maximum 2-edge path packing problem.

**Keywords:** Analysis of algorithms, maximum triangle packing, 2-edge paths.

## 1 Introduction

Let  $G = (V, E)$  be a complete (undirected) graph with vertex set  $V$  such that  $|V| = 3n$ , and edge set  $E$ . For  $e \in E$  let  $w(e) \geq 0$  be its weight. For  $E' \subseteq E$  we denote  $w(E') = \sum_{e \in E'} w(e)$ . For a random subset  $E' \subseteq E$ ,  $w(E')$  denotes the expected value. In this paper, a  $k$ -path is a simple  $k$ -edge path, and similarly a  $k$ -cycle is a  $k$ -edge simple cycle. A 3-cycle is also called a *triangle*. The MAXIMUM TRIANGLE PACKING PROBLEM is to compute a set of  $n$  vertex-disjoint triangles with maximum total edge weight.

In this paper we propose a randomized algorithm which is the first approximation algorithm with performance guarantee strictly better than 0.5. Specifically, our algorithm is an  $(\frac{89}{169} - \epsilon)$ -approximation for any given  $\epsilon > 0$ .

The MAXIMUM 2-EDGE PATH PACKING PROBLEM requires to compute a maximum weight set of vertex disjoint 2-edge paths. We improve the best known approximation bound for this problem and prove that our triangle packing algorithm is also a  $(\frac{35}{67} - \epsilon)$ -approximation algorithm for this problem.

## 2 Related Literature

The problems of whether the vertex set of a graph can be covered by vertex disjoint 2-edge paths or vertex disjoint triangles are NP-complete (see [6] p. 76 and 192, respectively). These results imply that the two problems defined in the introduction are NP-hard.

The problems considered in this paper are special cases of the 3-SET PACKING PROBLEM. In the unweighted version of this problem, a collection of sets of

cardinality at most 3 each, is given. The goal is to compute a maximum number of disjoint sets from this collection. In the weighted version, each set has a weight, and the goal is to find a sub-collection of disjoint sets having maximum total weight. We now survey the existing approximation results for the 3-SET PACKING PROBLEM, which of course also apply to the problems treated in this paper.

Hurkens and Schrijver [11] proved that a natural local search algorithm can be used to give a  $(\frac{2}{3} - \epsilon)$ -approximation algorithm for UNWEIGHTED 3-SET PACKING for any  $\epsilon > 0$  (see also [7]). Arkin and Hassin [1] analyzed the local search algorithm when applied to the WEIGHTED  $k$ -SET PACKING PROBLEM. They proved that for  $k = 3$ , the result is a  $(\frac{1}{2} - \epsilon)$ -approximation. Bafna, Narayanan, and Ravi [2] analyzed a restricted version of the local search algorithm in which the depth of the search is also  $k$ , for a more general problem of computing a maximum independent set in  $(k + 1)$ -claw free graphs. For  $k = 3$  it gives a  $\frac{3}{7}$ -approximation). A more involved algorithm for  $k$ -set packing, that combines greedy and local search ideas was given by Chandra and Halldórsson [4]. This algorithm yields improved bounds for  $k \geq 6$ . Finally, Berman [3] improved the approximation ratios for all  $k \geq 4$ , and for the maximum independent set in  $(k + 1)$ -claw free graphs, however, the bound for  $k = 3$  is still 0.5.

Kann [12] proved that the UNWEIGHTED TRIANGLE PACKING PROBLEM is APX-complete even for graphs with maximum degree 4. Chlebík and Chlebíková [5] proved that it is NP-hard to obtain an approximation factor better than 0.9929.

The MAXIMUM TRAVELING SALESMAN PROBLEM (MAX TSP) asks to compute in an edge weighted graph a Hamiltonian cycle (or *tour*) of maximum weight. MAX TSP is a relaxation of MAX  $k$ -EDGE PATH PACKING for any  $k$ . Therefore, an  $\alpha$ -approximation algorithm for the former problem can be used to approximate the latter [10]: simply delete ever  $(k + 1)$ -st edge from the TSP solution. Choose the starting point so that the deleted weight is at most  $\frac{1}{k+1}$  the total weight. The result is an  $(\alpha \frac{k}{k+1})$ -approximation. By applying the  $\frac{25}{33}$ -approximation randomized algorithm of Hassin and Rubinstein [9] for MAX TSP, we obtain a  $(\frac{50}{99} - \epsilon)$ -approximation for  $k = 2$ . For the MAXIMUM 3-EDGE PATH PACKING PROBLEM there is a better  $\frac{3}{4}$  bound in [10].

In this paper we give the first algorithm whose performance guarantee is strictly greater than  $\frac{1}{2}$  for MAXIMUM TRIANGLE PACKING. Our bound is  $(\frac{89}{169} - \epsilon)$  for every  $\epsilon > 0$ . The algorithm is described in Section 3. We also improve the above-mentioned bound for MAXIMUM WEIGHTED 2-PATH PACKING. This result is described in Section 4. Specifically, our algorithm returns a  $(\frac{35}{67} - \epsilon)$ -approximation for this problem.

### 3 Maximum Weighted Triangle Packing

A (*perfect*) *binary 2-matching* (also called *2-factor* or *cycle cover*) is a subgraph in which every vertex in  $V$  has a degree of exactly 2. A *maximum binary 2-matching* is one with maximum total edge weight. Hartvigsen [8] showed how

to compute a maximum binary 2-matching in  $O(n^3)$  time (see [14] for another  $O(n^2|E|)$  algorithm). Note that a 2-matching consists of disjoint simple cycles of at least three edges each, that together span  $V$ .

We denote the weight of an optimal solution by  $opt$ .

Algorithm *WTP* is given in Figure 1. The algorithm starts by computing a maximum binary 2-matching. Long cycles, where  $|C| > \epsilon^{-1}$  ( $|C|$  denotes the number of vertices of  $C$ ), are broken into paths with at most  $\epsilon^{-1}$  edges each, loosing a fraction of at most  $\epsilon$  of their weight. To simplify the exposition, the algorithm completes the paths into cycles to form a cycle cover  $\mathcal{C}$ . The cycle cover  $\mathcal{C}$  consists of vertex disjoint cycles  $C_1, \dots, C_r$  satisfying  $3 \leq |C_i| \leq \epsilon^{-1} + 1$   $i = 1, \dots, r$ . Since the MAXIMUM CYCLE COVER PROBLEM is a relaxation of the WEIGHTED MAXIMUM TRIANGLE PACKING PROBLEM,

$$w(\mathcal{C}) \geq (1 - \epsilon)opt.$$

Algorithm *WTP* constructs three solutions and selects the best one. The first solution is attractive when a large fraction of  $opt$  comes from edges that belong to  $\mathcal{C}$ . The second is attractive when the a large fraction of  $opt$  comes from edges whose two vertices are on the same cycle of  $\mathcal{C}$ . The third solution deals with the remaining case, in which the optimal solution also uses a considerable weight of edges that connect distinct cycles of  $\mathcal{C}$ .

*WTP*

**input**

1. A complete undirected graph  $G = (V, E)$  with weights  $w_{ij}$  ( $i, j \in E$ ).

2. A constant  $\epsilon > 0$ .

**returns** A triangle packing.

**begin**

Compute a maximum cycle cover  $\{C'_1, \dots, C'_{r'}\}$ .

**for**  $i = 1, \dots, r'$ :

**if**  $|C'_i| > \epsilon^{-1}$

**then**

            Remove from  $C'_i$   $\lceil \epsilon |C'_i| \rceil$  edges of total weight at most  $\frac{\lceil \epsilon |C'_i| \rceil}{|C'_i|} w(C'_i)$ ,

            to form a set of paths of at most  $\epsilon^{-1}$  edges each.

            Close each of the paths into a cycle by adding an edge.

**end if**

**end for**

$\mathcal{C} = \{C_1, \dots, C_r\} :=$  the resulting cycle cover.

$TP_1 := A1(G, \mathcal{C})$ .

$TP_2 := A2(G, \mathcal{C})$ .

$TP_3 := A3(G, \mathcal{C})$ .

**return** the solution with maximum weight among  $TP_1, TP_2$  and  $TP_3$ .

**end** *WTP*

**Fig. 1.** Algorithm *WTP*

A1

**input**

1. A complete undirected graph  $G = (V, E)$  with weights  $w_{ij}$  ( $i, j \in E$ ).
2. A cycle cover  $\mathcal{C} = \{C_1, \dots, C_r\}$ .

**returns** A triangle packing  $TP_1$ .**begin** $TP_1 := \emptyset$ .**for**  $i = 1, \dots, r$ :    **if**  $|C_i| = 3$         **then**             $TP_1 := TP_1 \cup \{C_i\}$ .    **elseif**  $|C_i| = 5$         **then**            Let  $e_1, \dots, e_5$  be the edges of  $C_i$  in cyclic order.             $\hat{w}_i := w_i + w_{i+1} + \frac{1}{2}w_{i+3}$  [indices taken mod 2].             $\hat{w}_j = \max\{\hat{w}_i : i = 1, \dots, 5\}$ .             $TP_1 := TP_1 \cup \{j, j+1, j+2\}$ .             $E' := E' \cup \{j+3, j+4\}$ .    **elseif**  $|C_i| \notin \{3, 5\}$         **then**            Let  $e_1, \dots, e_c$  be the edges of  $C_i$  in cyclic order.            Add to  $TP_1$  triangles induced by adjacent pairs of edges from  $C_i$ , of total weight at least  $\frac{w(C_i)}{2}$ .            Add to  $V'$  the vertices of  $C_i$  that are not incident to selected triangles.        **end if****end for** $E'' := \left\lceil \frac{|E'|}{2} \right\rceil$  heaviest edges from  $E'$ .**for every**  $e \in E''$     Add to  $TP_1$  triangles formed from  $e$  and a third vertex either from  $E' \setminus E''$  or from  $V'$ .**return**  $TP_1$ .**end** A1**Fig. 2.** Algorithm A1

**The first solution** is constructed by Algorithm A1 (see Figure 2). It selects all the 3-cycles of  $\mathcal{C}$ . From every  $k$ -cycle  $C_i$  with  $k \neq 3, 5$ , the algorithm selects 3-sets consisting of vertices of pairs of adjacent edges in  $C_i$  of total weight at least  $\frac{1}{2}w(C_i)$ , to form triangles in the solution. This task can be done by selecting an appropriate edge of the cycle, and then deleting this edge with one or two of its neighboring edges (depending on the value of  $|C_i| \bmod 3$ ), and then every third edge of the cycle according to an arbitrary orientation.

5-cycles obtain special treatment (since the above process would only guarantee  $\frac{2}{5}$  of their weight): From each 5-cycle we select three edges. Two of these

A2

**input**1. A complete undirected graph  $G = (V, E)$  with weights  $w_{ij}$  ( $i, j \in E$ ).2. A cycle cover  $\mathcal{C} = \{C_1, \dots, C_r\}$ .**returns** A triangle packing  $TP_2$ .**begin** $V_1, \dots, V_r :=$  the vertex sets of  $C_1, \dots, C_r$ , respectively.**for**  $j = 0, \dots, n$ : $F(0, j) = 0$ .**end for****for**  $i = 1, \dots, r$ : $F(i, 0) := 0$ .**for**  $j = 1, \dots, n$ : $W(i, j) :=$  maximum weight of at most  $j$  disjoint 3-sets and 2-sets contained in  $V_i$ .**end for** $F(i, j) := \max\{W(i, k) + F(i-1, j-k) : k \leq j\}$ .**end for** $P(r, n) :=$  a collection of 2-sets and 3-sets that gives  $F(r, n)$ .**return**  $TP_2$ , a completion of  $P(r, n)$  into a triangle packing.**end A2****Fig. 3.** Algorithm A2

edges are adjacent and the third is disjoint from the two. The former edges define a triangle which is added to the solution. The third edge is added to a special set  $E'$ . The selection is made so that the weight of the adjacent pair plus half the weight of the third pair is maximized. After going through every cycle, a subset of  $E'$  of total weight at least  $\frac{1}{2}w(E')$  is matched to unused vertices to form triangles. (Since  $|V| = 3n$ , there should be a sufficient number of unused vertices.) Thus, in total we gain at least of the cycle's weight.

**The second solution** is constructed by Algorithm A2 (see Figure 3). The algorithm enumerates all the possible packings of vertex disjoint 2-sets and 3-sets in the subgraph induced by the vertex set of each cycle  $C_i \in \mathcal{C}$ . The weight of a subset is defined to be the total edge weight of the induced subgraph (a triangle or a single edge). A dynamic programming recursion is then used to compute the maximum weight of  $n$  subsets from the collection. In this formulation, we use  $W(i, j)$  to denote the maximum weight that can be obtained from at most  $j$  such subsets, subject to the condition that the vertex set of each subset must be fully contained in the vertex set of one of the cycles  $C_1, \dots, C_i$ . After computing  $W(r, n)$ , the solution that produced this value is completed in an arbitrary way to a triangle packing  $TP_2$ .

**The third solution** is constructed by Algorithm A3 (see Figure 5). It starts by deleting edges from  $\mathcal{C}$  according to Procedure *Delete* described in Figure 4.



*Delete*

**input** A set of cycles  $\mathcal{C} = \{C_1, \dots, C_r\}$ .

**returns** A set of paths  $\mathcal{P}$ .

**begin**

**for**  $i = 1, \dots, r$ :

Randomly select an edge from  $C_i$  and mark it  $e_1$ .

Delete  $e_1$ .

Denote the edges of  $C_i$  in cyclic order according to an arbitrary orientation and starting at  $e_1$  by  $e_1, \dots, e_c$ , where  $c = |C_i| = 4k + l$  and  $l \in \{0, \dots, 3\}$ .

Delete from  $C_i$  the edges  $e_j$  such that  $j \equiv 1 \pmod{4}$  and  $j \leq c - 3$ .

**if**  $l = 1$

**then**

Delete  $e_{c-1}$  with probability  $\frac{1}{4}$ .

**elseif**  $l = 2$

**then**

Delete  $e_{c-1}$  with probability  $\frac{1}{2}$ .

**elseif**  $l = 3$  and  $c > 3$

**then**

Delete  $e_{c-2}$  with probability  $\frac{3}{4}$ .

**end if**

**end for**

Denote the resulting path set by  $\mathcal{P}$ .

**return**  $\mathcal{P}$ .

**end** *Delete*

**Fig. 4.** Procedure *Delete*

The result is a collection  $\mathcal{P}$  of subpaths of  $\mathcal{C}$  such that the following lemma holds:

**Lemma 1.** Consider a cycle  $C_i \in \mathcal{C}$ . Let  $E_d^i$  be the edge set deleted from  $C_i$  by Procedure *Delete*. Then:

1.  $E_d^i \neq \emptyset$ .
2. The edges in  $E_d^i$  are vertex disjoint.
3. The expected size of  $E_d^i$  is at least  $\frac{1}{4}|C_i|$ .
4. The probability that any given vertex of  $C_i$  is adjacent to an edge in  $E_d^i$  is at least  $\frac{1}{2}$ .

*Proof.*

1. The first property holds since an edge  $e_1$  is deleted from the  $C_i$ .
2. The second property holds by the way edges are selected for deletion.
3. The expected value of  $|E_d^i|$  is  $\frac{1}{3}|C_i|$  for a triangle and  $\frac{1}{4}|C_i|$  otherwise.
4. If a vertex belongs to a 3-cycle in  $\mathcal{C}$  then it is incident to  $e_1$  with probability  $\frac{2}{3}$ . If the vertex is in a  $k$ -cycle,  $k > 3$ , then since the expected size of  $E_d^i$

A3

**input**

1. A complete undirected graph  $G = (V, E)$ ,  $|V| = 3n$ , with weights  $w_{ij}$   $(i, j) \in E$ .
2. A cycle cover  $\mathcal{C} = \{C_1, \dots, C_r\}$ .

**returns** A triangle packing  $TP_3$ .

**begin**

Let  $E'$  be the edges of  $G$  with two ends in different cycles of  $\mathcal{C}$ .

Compute a maximum weight matching  $M' \subset E'$ .

$\mathcal{P} := \text{Delete}(\mathcal{C})$ .

$M := \{(i, j) \in M' : i \text{ and } j \text{ are end vertices of paths in } \mathcal{P}\}$ .

%  $M \cup \mathcal{P}$  consists of paths  $P_1^*, \dots, P_s^*$  and cycles  $C_1^*, \dots, C_t^*$  such that each cycle contains at least two edges from  $M$ .%

$\mathcal{P}^* := \{P_1^*, \dots, P_s^*\}$ .

**begin cycle canceling step:**

**for**  $i = 1, \dots, t$ :

    Randomly select an edge  $e \in C_i^* \cap M$ .

    Add  $C_i^* \setminus \{e\}$  to  $\mathcal{P}^*$ .

**end for**

**end cycle canceling step**

Arbitrarily complete  $\mathcal{P}^*$  to a Hamiltonian cycle  $T$ .

Delete  $n$  edges from  $T$  to obtain a collection  $\mathcal{P}_3$  of 2-edge paths of total weight at least  $\frac{2}{3}w(T)$ .

**return**  $TP_3$ , a completion of  $\mathcal{P}_3$  into a triangle packing.

**end** A3

**Fig. 5.** Algorithm A3

is  $\frac{|C_i|}{4}$  and these edges are vertex disjoint, the expected number of vertices incident to these edges is  $\frac{|C_i|}{2}$ . □

Consider a given cycle  $C \in \mathcal{C}$  with  $|C| = 4k + l$ . If  $C$  is a triangle then exactly one edge is deleted by Procedure *Delete*. If  $l = 0$  then every fourth edge is deleted. If  $l \in \{1, 2, 3\}$  and  $|C| > 3$  then the number of deleted edges may be  $k$  or  $k + 1$ , and in each case, their location relative to  $e_1$  is uniquely determined. Let us define a binary random variable  $X_C$  such that the number of deleted edges in  $C$  is  $k + X_C$ .  $X_C$  uniquely determines the *deletion pattern* in  $C$ , specifying the spaces among deleted edges, but not their specific location. Since every edge has equal probability to be chosen as  $e_1$ , the possible mappings of the deletion pattern into  $C$  have equal probabilities. Suppose that the deletion pattern is known. Every mapping of it into  $C$  specifies a subset  $S$  of vertices that are incident to deleted edges, and thus these vertices are the end vertices of the paths in  $\mathcal{P}$ . We call these vertices *free*.

Algorithm A3 computes a maximum matching  $M'$  over the set  $E' \subset E$  of edges with ends in distinct vertices. Only the subset  $M \subseteq M'$  of edges whose two ends become free in the deletion procedure is useful, and it is used to generate paths with the undeleted edges from  $\mathcal{C}$ . However, cycles may also be generated

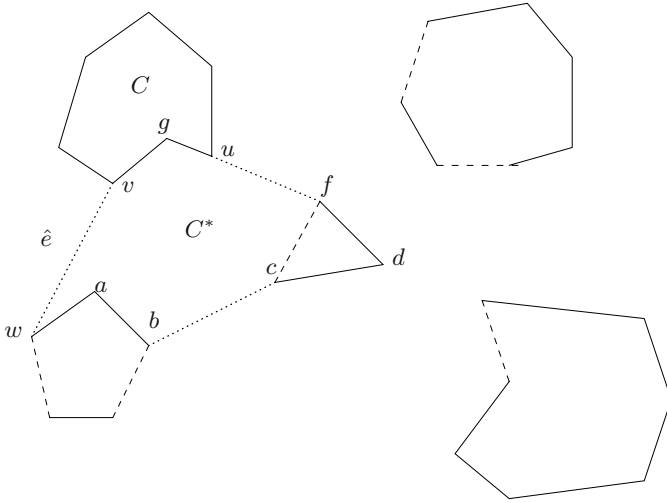
this way, and a cycle canceling step further deletes edges to cancel these cycles. Figure 6 illustrates this situation, where broken lines denote deleted edges and dotted lines denote edges that belong to  $M$ . The cycle  $C^*$  consists of the vertices  $(v, w, a, b, c, d, f, u, g)$ , and one of the edges  $(v, w)$ ,  $(b, c)$  and  $(f, u)$  that belong to  $M$  will be deleted to cancel  $C^*$ .

Consider an edge  $\hat{e} = (v, w) \in M$ . Denote by  $\hat{\pi}$  the probability that  $\hat{e}$  is deleted in the cycle canceling step, given that its vertices  $v$  and  $w$  are free (i.e.,  $v, w \in S$ ).

**Lemma 2.**  $\hat{\pi} \leq \frac{1}{4}$ .

*Proof.* Suppose that  $v \in C$ . For every  $u \in C$ , define  $\text{dist}(u, v)$  to be the minimum (over the two possibilities) number of edges on a  $u-v$  subpath of  $C$ . (For example,  $\text{dist}(u, v) = 2$  in Figure 6.)

To prove that  $\hat{\pi} < \frac{1}{4}$ , we define the following events:



**Fig. 6.** A cycle in  $M \cup \mathcal{P}$

$E_{\delta, x}$ :  $X_C = x$ ;  $v, w \in S$ ;  $\mathcal{P} \cup M$  contains a  $u-v$  path  $P$  such that  $\hat{e}$  is on  $P$ ,  $C \cap P = \{u, v\}$ ,  $u \in S$ , and  $\text{dist}(u, v) = \delta$ . (For example,  $P = (v, w, a, b, c, d, f, u)$  and  $\delta = 2$  in Figure 6.)

$E_C$ :  $u$  and  $v$  belong to a *common* path in  $\mathcal{P}$ .

$E_D$ :  $u$  and  $v$  belong to *different* paths in  $\mathcal{P}$ .

If  $\hat{e}$  belongs to a cycle  $C^* \subseteq \mathcal{P} \cup M$ , then in the event  $E_C$ ,  $|C^* \cap M| \geq 2$ , whereas in the event  $E_D$ ,  $|C^* \cap M| \geq 4$ . Therefore the respective probabilities that  $\hat{e}$  is deleted in the cycle canceling step are at most  $\frac{1}{2}$  and  $\frac{1}{4}$ . Using this observation,

$$\hat{\pi} \leq \sum_{\delta=1}^{\lfloor \frac{|C|}{2} \rfloor} \sum_{x=0}^1 Pr(E_{\delta,x}) \left( \frac{1}{2} Pr(E_C | E_{\delta,x}) + \frac{1}{4} Pr(E_D | E_{\delta,x}) \right).$$

The proof is completed by Lemma 3 below.  $\square$

Let  $\pi(\delta, x) = \frac{1}{2} Pr(E_C | E_{\delta,x}) + \frac{1}{4} Pr(E_D | E_{\delta,x})$ .

**Lemma 3.** *For every  $\delta$  and  $x$ ,*

$$\pi(\delta, x) \leq \frac{1}{4}.$$

*Proof.* The proof contains a detailed case analysis. It will be included in the full version of this paper.  $\square$

**Theorem 1.**  $\max\{w(TP_1), w(TP_2), w(TP_3)\} \geq \frac{89}{169}(1 - \epsilon)opt.$

*Proof.* Algorithm *WTP* first breaks long cycles loosing at most a fraction  $\epsilon$  of their weight, and obtains a cycle cover  $\mathcal{C}$ . Thus,  $w(\mathcal{C}) \geq (1 - \epsilon)opt$ .

Let  $\alpha$  denote the proportion of  $w(\mathcal{C})$  contained in 3-edge cycles. The solution  $TP_1$  contains all the triangles of  $\mathcal{C}$  and at least half of the weight of any other cycle. Thus,

$$w(TP_1) \geq \left( \alpha + \frac{1 - \alpha}{2} \right) w(\mathcal{C}) = \frac{1 + \alpha}{2} w(\mathcal{C}) \geq \frac{1 + \alpha}{2} (1 - \epsilon)opt. \quad (1)$$

Consider an optimal solution. Define  $T_{int}$  to be the edges of this solution whose end vertices are in the same connectivity component of  $\mathcal{C}$ , and suppose that  $w(T_{int}) = \beta opt$ . Then

$$w(TP_2) \geq w(T_{int}) = \beta opt. \quad (2)$$

Algorithm *A3* computes a Hamiltonian path  $T$ , and then constructs a triangle packing of weight at least  $\frac{2}{3}w(T)$ .  $T$  is built as follows: First  $\frac{1}{3}$  of the weight of any triangle, and  $\frac{1}{4}$  of any other cycle of  $\mathcal{C}$  is deleted, leaving weight of  $\left[ \frac{2}{3}\alpha + \frac{3}{4}(1 - \alpha) \right] w(\mathcal{C})$ . Then edges from the matching  $M'$  are added. Originally  $w(M') \geq \frac{1-\beta}{2}opt$ . However only edges with two free ends are used, and by Lemma 1(4) their expected weight is at least  $\frac{1}{4}w(M') \geq \frac{1-\beta}{8}opt$ . Then cycles are broken, deleting at most  $\frac{1}{4}$  of the remaining weight (by Lemma 2). Hence the added weight is at least  $\frac{3}{32}(1 - \beta)$ . Altogether,

$$\begin{aligned} w(T) &\geq \left[ \frac{2}{3}\alpha + \frac{3}{4}(1 - \alpha) \right] w(\mathcal{C}) + \frac{3}{32}(1 - \beta)opt \\ &\geq \left[ \frac{2}{3}\alpha + \frac{3}{4}(1 - \alpha) \right] (1 - \epsilon)opt + \frac{3}{32}(1 - \beta)opt. \end{aligned}$$

From this, a solution is formed after deleting at most  $\frac{1}{3}$  of the weight. Hence,

$$w(TP_3) \geq \left( \frac{9}{16} - \frac{1}{18}\alpha - \frac{1}{16}\beta \right) (1 - \epsilon)opt. \quad (3)$$

It is now easy to prove that  $\max\{w(TP_1), w(TP_2), w(TP_3)\} \geq \frac{89}{169}(1 - \epsilon)opt$ : If  $\alpha < \frac{89}{169}$  then  $w(TP_1) > \frac{89}{169}(1 - \epsilon)opt$ , if  $\beta < \frac{89}{169}$  then  $w(TP_2) > \frac{89}{169}(1 - \epsilon)opt$ , and if these conditions do not hold then  $w(TP_3) > \frac{89}{169}(1 - \epsilon)opt$ .  $\square$

The time consuming parts of the algorithm are the computation of a maximum 2-matching and the dynamic program in Algorithm A2. The first can be computed in time  $O(n^3)$  as in [8]. Since the latter is executed on cycles with at most  $\epsilon^{-1}$  edges each, it also takes  $O(n^3)$  for any constant  $\epsilon$ .

## 4 2-Path Packing

Consider now the MAXIMUM 2-PATH PACKING PROBLEM. We apply Algorithm WTP, with two slight changes: one is that we do not complete 2-paths into triangles, and the other is that in Algorithm A1 we select from every triangle of  $\mathcal{C}$  the two longest edges to form a 2-path. The analysis is identical, except for that the bound guaranteed by Algorithm A1 is only  $w(TP_1) \geq [\frac{2}{3}\alpha + \frac{1}{2}(1 - \alpha)] w(\mathcal{C})$ . The resulting approximation bound is  $\frac{35}{67} - \epsilon$ .

## References

1. E. Arkin and R. Hassin, "On local search for weighted packing problems," *Mathematics of Operations Research* **23** (1998), 640-648.
2. V. Bafna, B. Narayanan, and R. Ravi, "Nonoverlapping local alignments (weighted independent sets of axis-parallel rectangles)," *Discrete Applied Mathematics* **71** (1996), 41-53.
3. P. Berman, "A  $d/2$  approximation for maximum weight independent set in  $d$ -claw free graphs," *Nordic Journal of Computing* **7** (2000), 178-184.
4. B. Chandra and M. M. Halldórsson, "Greedy local improvement and weighted set packing approximation," *Journal of Algorithms* **39** (2001), 223-240.
5. M. Chlebík and J. Chlebíková, "Approximating hardness for small occurrence instances of NP-hard problems," in *Algorithms and Complexity*, R. Petreschi, G. Parsiano, and R. Silvestri (Eds.) CIAC 2003, *LNCS* **2653** 152-164.
6. M. R. Garey, and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1978.
7. M. M. Halldórsson, "Approximating discrete collections via local improvements," *Proceedings of the Sixth SODA 95. Annual ACM-SIAM Symposium on Discrete Algorithms* SODA'95, 160-169, 1995.
8. D. Hartvigsen, *Extensions of Matching Theory*. Ph.D. Thesis, Carnegie-Mellon University (1984).
9. R. Hassin and S. Rubinstein, "Better approximations for Max TSP," *Information Processing Letters* **75** (2000), 181-186.

10. R. Hassin and S. Rubinstein, "An approximation algorithm for maximum packing of 3-edge paths," *Information Processing Letters*, **63** (1997) 63-67.
11. C.A.J. Hurkens and A. Schrijver, "On the size of systems of sets every  $t$  of which have an SDR, with an application to the worst-case ratio of heuristics for packing problems," *SIAM J. on Discrete Mathematics* **2**, (1989) 68-72.
12. V. Kann, "Maximum bounded 3-dimensional matching is MAX SNP-complete," *Information processing Letters* **37**, 1991, 27-35.
13. B. Korte and D. Hausmann, "An analysis of the greedy heuristic for independence systems," *Annals of Discrete Mathematics* **2**, 1978, 65-74.
14. J. F. Pekny and D. L. Miller, "A staged primal-dual algorithm for finding a minimum cost perfect two-matching in an undirected graph", *ORSA Journal on Computing* **6** (1994), 68-81.

# Approximate Parameterized Matching

Carmit Hazay<sup>1</sup>, Moshe Lewenstein<sup>1</sup>, and Dina Sokol<sup>2</sup>

<sup>1</sup> Bar Ilan University  
`{harelc,moshe}@cs.biu.ac.il`  
<sup>2</sup> Brooklyn College of CUNY  
`sokol@sci.brooklyn.cuny.edu`

**Abstract.** Two equal length strings  $s$  and  $s'$ , over alphabets  $\Sigma_s$  and  $\Sigma_{s'}$ , *parameterize match* if there exists a bijection  $\pi : \Sigma_s \rightarrow \Sigma_{s'}$ , such that  $\pi(s) = s'$ , where  $\pi(s)$  is the renaming of each character of  $s$  via  $\pi$ . *Approximate parameterized matching* is the problem of finding for a pattern  $p$ , at each location of a text string  $t$ , a bijection  $\pi$  that maximizes the number of characters that are mapped from  $p$  to the appropriate  $|p|$ -length substring of  $t$ .

Our main result is an  $O(nk^{1.5} + mk \log m)$  time algorithm for this problem where  $m = |p|$  and  $n = |t|$ .

## 1 Introduction

In the traditional pattern matching model one seeks exact occurrences of a given pattern  $p$  in a text  $t$ , i.e. text locations where every text symbol is *equal* to its corresponding pattern symbol. For two equal length strings  $s$  and  $s'$  we say that  $s$  is a *parameterized match*, *p-match* for short, of  $s'$  if there exists a bijection  $\pi$  from the alphabet of  $s$  to the alphabet of  $s'$  such that every symbol of  $s'$  is equal to the *image under  $\pi$*  of the corresponding symbol of  $s$ . In the *parameterized matching* problem, introduced by Baker [6,8], one seeks text locations for which the pattern  $p$  *p-matches* the substring of length  $|p|$  beginning at that location.

Baker introduced parameterized matching for applications that arise in software tools for analyzing source code. It turns out that the parameterized matching problem arises in other applications such as in image processing and computational biology, see [2].

In [6,8] an optimal, linear time algorithm was given for p-matching. However, it was assumed that the alphabet was of constant size. Amir et.al. [4] presented tight bounds for p-matching in the presence of an unbounded size alphabet.

In [7] a novel method was presented for parameterized matching by constructing *parameterized suffix trees*, which also allows for online p-matching. The parameterized suffix trees are constructed by converting the pattern string into a *predecessor string*. A *predecessor string* of a string  $s$  has at each location  $i$  the distance between  $i$  and the location containing the previous appearance of the

symbol. The first appearance of each symbol replaced with a 0. For example, the predecessor string of *aabbaba* is 0, 1, 0, 1, 3, 2, 2. A simple and well-known fact is that:

**Lemma 1.**  *$s$  and  $s'$   $p$ -match iff they have the same predecessor string.*

The parameterized suffix tree was further explored by Kosaraju [15] and faster constructions were given by Cole and Hariharan [10].

One of the interesting problems in web searching is searching for color images, see [1,19,21]. The simplest possible case is searching for an icon in a screen, a task that the Human-Computer Interaction Lab at the University of Maryland was confronted with. If the colors are fixed, this is exact 2-dimensional pattern matching [3]. However, if the color maps in pattern and text differ, the exact matching algorithm would not find the pattern. Parameterized 2-dimensional search is precisely what is needed. A nearly optimal algorithm to solve the 2-dimensional parameterized matching problem was given in [2].

In reality, when searching for an image, one needs to take into account the occurrence of errors that distort the image. Errors occur in various forms, depending upon the application. The two most classical distance metrics are the Hamming distance and the edit distance [18].

In [9] the parameterized match problem was considered in conjunction with the edit distance. Here the definition of edit distance was slightly modified so that the edit operations are defined to be insertion, deletion and parameterized replacements, i.e. the replacement of a substring with a string that  $p$ -matches it. An algorithm for finding the “ $p$ -edit distance” of two strings was devised whose efficiency is close to the efficiency of the algorithms for computing the classical edit distance. Also an algorithm was suggested for the decision variant of the problem where a parameter  $k$  is given and it is necessary to decide whether the strings are within ( $p$ -edit) distance  $k$ .

However, it turns out that the operation of parameterized replacement relaxes the problem to an easier problem. A more rigid, but more realistic, definition for the Hamming distance variant was given in [5]. For a pair of equal length strings  $s$  and  $s'$  and a bijection  $\pi$  defined on the alphabet of  $s$ , the  $\pi$ -mismatch is the Hamming distance between the image under  $\pi$  of  $s$  and  $s'$ . The minimal  $\pi$ -mismatch over all bijections  $\pi$  is the approximate parameterized match. The problem considered in [5] is to find for each location  $i$  of a text  $t$  the approximate parameterized match of a pattern  $p$  with the substring beginning at location  $i$ . In [5] the problem was defined and linear time algorithms were given for the case where the pattern is binary or the text is binary. However, this solution does not carry over to larger alphabets.

Unfortunately, under this definition the methods for classical string matching with errors for Hamming distance, e.g. [14,17], also known as pattern matching



with mismatches, seem to fail. For example the method in [14] uses a suffix tree and precomputed longest common ancestor (LCA) information.

When attempting to apply this technique using a parameterized suffix tree, it fails. This fails because separate parameterized matches of substrings do not necessarily add up to a parameterized match. See the example below (substring 1-4 and substring 6-9). Different  $\pi$ 's are require for these p-matches. This also emphasizes why the definition of [9] is a simplification.

$$\begin{aligned} p &= \quad a \ b \ a \ b \ a \ a \ b \ a \ a \ \dots \\ t &= \quad \dots c \ d \ c \ d \ d \ e \ f \ e \ e \ \dots \end{aligned}$$

In this paper we consider the problem of *parameterized matching with  $k$  mismatches*. The parameterized matching problem with  $k$  mismatches seeks all *p-matches* of a pattern  $p$  in a text  $t$ , with at most  $k$  mismatches.

For the case where  $|p|=|t|=m$ , which we call the string comparison problem we show that this problem is in a sense equivalent to the maximum matching problem on graphs. An  $O(m^{1.5})$  algorithm for the problem follows, by using maximum matching algorithms. A matching "lower bound" follows, in the sense that an  $o(m^{1.5})$  algorithm implies better algorithms for maximum matching. For the string comparison problem with threshold  $k$ , we show an  $O(m + k^{1.5})$  time algorithm and a "lower bound" of  $\Omega(m + \frac{k^{1.5}}{\log m})$ .

The main result of the paper is an algorithm that solves the parameterized matching problem with  $k$  mismatches problem, given a pattern of length  $m$  and a text of length  $n$ , in  $O(nk^{1.5} + mk \log m)$  time. This immediately yields a 2-dimensional algorithm of time complexity  $O(n^2mk^{1.5} + m^2k \log m)$ , where  $|p|=m^2$  and  $|t|=n^2$ .

**Roadmap:** In section 2 we give preliminaries and definitions of the problem. In section 3 we present an algorithm for the string comparison problem. We also reduce maximum matching to the string comparison problem. In section 4 we present an algorithm for the parameterized matching with  $k$  mismatches.

## 2 Preliminaries and Definitions

Given a *string*  $s = s_1 s_2 \dots s_n$  of *length*  $|s| = n$  over an alphabet  $\Sigma_s$ , and a bijection  $\pi$  from  $\Sigma_s$  to some other alphabet  $\Sigma'_s$ , the *image* of  $s$  under  $\pi$  is the string  $s' = \pi(s) = \pi(s_1) \cdot \dots \cdot \pi(s_n)$ , that is obtained by applying  $\pi$  to all characters of  $s$ .

Given two equal-length strings  $w$  and  $u$  over alphabets  $\Sigma_w$  and  $\Sigma_u$  and a bijection  $\pi$  from  $\Sigma_w$  to  $\Sigma_u$  the  $\pi$ -mismatch between  $w$  and  $u$  is  $Ham(\pi(w), u)$ , where  $Ham(s, t)$  is the Hamming distance between  $s$  and  $t$ . We say that  $w$  *parameterized  $k$ -matches*  $u$  if there exists a  $\pi$  such that the  $\pi$ -mismatch  $\leq k$ . The

*approximate parameterized match* between  $w$  and  $u$  is the minimum  $\pi$ -mismatch (over all bijections  $\pi$ ).

For given input text  $x$ ,  $|x| = n$ , and pattern  $y$ ,  $|y| = m$ , the problem of *approximate parameterized matching* for  $y$  in  $x$  consists of computing the *approximate parameterized match* between  $y$  and every (consecutive) substring of length  $m$  of  $x$ . Hence, approximate parameterized searching requires computing the  $\pi$  yielding minimum  $\pi$ -mismatch for  $y$  at each position of  $x$ . Of course, the best  $\pi$  is not necessarily the same at every position. The problem of *parameterized matching with  $k$  mismatches* consists of computing for each location  $i$  of  $x$  whether  $y$  parameterized  $k$ -matches the (consecutive) substring of length  $m$  beginning at location  $i$ . Sometimes  $\pi$  itself is also desired (however, here any  $\pi$  with  $\pi$ -mismatch of no more than  $k$  will be satisfactory).

### 3 String Comparison Problem

We begin by evaluating two equal-length strings for a parameterized  $k$ -match as follows:

- **Input:** Two strings,  $s = s_1, s_2, \dots, s_m$  and  $s' = s'_1, s'_2, \dots, s'_m$ , and an integer  $k$ .
- **Output:** True, if there exists  $\pi$  such that the  $\pi$ -mismatch of  $s$  and  $t$  is  $\leq k$ .

The naive method to solve this problem is to check the  $\pi$ -mismatch for every possible  $\pi$ . However, this takes exponential time.

A slightly more clever way to solve the problem is to reduce the problem to a maximal bipartite weighted matching in a graph. We construct a bipartite graph  $B = (U \cup V, E)$  in the following way.  $U = \Sigma_s$  and  $V = \Sigma_{s'}$ . There is an edge between  $a \in \Sigma_s$  and  $b \in \Sigma_{s'}$  iff there is at least one  $a$  in  $s$  that is aligned with a  $b$  in  $s'$ . The weight of this edge is the number of  $a$ 's in  $s$  that are aligned to  $b$ 's in  $s'$ . It is easy to see that:

**Lemma 2.** *A maximal weighted matching in  $B$  corresponds to a minimal  $\pi$ -mismatch, where  $\pi$  is defined by the edges of the matching.*

The problem of maximum bipartite matching has been widely studied and efficient algorithms have been devised, e.g. [11,12,13,20]. For integer weights, where the largest weight is bounded by  $|V|$ , the fastest algorithm runs in time  $O(E\sqrt{V})$  [20]. This solution yields an  $O(m^{1.5})$  time algorithm for the problem of parameterized string comparison with mismatches. The advantage of the algorithm is that it views the whole string at once and efficiently finds the best function. In general, for the parameterized pattern matching problem with inputs  $t$  of length  $n$ , and  $p$  of length  $m$ , the bipartite matching solution yields an  $O(nm^{1.5})$  time algorithm.

### 3.1 Reducing Maximum Matching to the String Comparison Problem

**Lemma 3.** *Let  $B = (U \cup V, E)$  be a bipartite graph. If the string comparison problem can be solved in  $O(f(m))$  time then a maximum matching in  $B$  can be found in  $O(f(|E|))$  time.*

Since it is always true that  $|E| < |V|^2$  it follows that  $|E|^{1/4} < \sqrt{|V|}$ . Hence, it would be surprising to solve the string comparison problem in  $o(m^{1.25})$ . In fact, even for graphs with a linear number of edges the best known algorithm is  $O(E\sqrt{V})$  and hence an algorithm for the string comparison problem in  $o(m^{1.5})$  would have implications on the maximum matching problem as well.

Note that for the string comparison problem one can reduce the approximate parameterized matching version to the parameterized matching with  $k$  mismatches version. This is done by binary searching on  $k$  to find the optimal solution for the string comparison problem in the approximate parameterized matching version. Hence an algorithm for detecting whether the string comparison problem has a bijection with fewer than  $k$  mismatches with time  $o(\frac{k^{1.5}}{\log k} + m)$  would imply a faster maximum matching algorithm.

### 3.2 Mismatch Pairs and Parameterized Properties

Obviously, one may use the solution of approximate parameterized string matching to solve the problem of parameterized matching with  $k$  mismatches. However, it is desirable to construct an algorithm with running time dependant on  $k$  rather than  $m$ . The bipartite matching method does not seem to give insight into how to achieve this goal. In order to give an algorithm dependent on  $k$  we introduce a new method for detecting whether two equal-length strings have a parameterized  $k$ -match. The ideas used in this new comparison algorithm will be used in the following section to yield an efficient solution for the problem of parameterized matching with  $k$  mismatches.

When faced with the problem of  $p$ -matching with mismatches, the initial difficulty is to decide, for a given location, whether it is a match or a mismatch. Simple comparisons do not suffice, since any symbol can match (or mismatch) every other symbol. In the bipartite solution, this difficulty is overcome by viewing the entire string at once. A bijection is found, over all characters, excluding at most  $k$  locations. Thus, it is obvious that the locations that are excluded from the matching are “mismatched” locations. For our purposes, we would like to be able to decide locally, *i.e. before  $\pi$  is ascertained*, whether a given location is “good” or “bad.”

**Definition 1 (mismatch pair).** *Let  $s$  and  $s'$  be two equal-length strings then a mismatch pair between  $s$  and  $s'$  is a pair of locations  $(i, j)$  such that one of the following holds,*

1.  $s_i = s_j$  and  $s'_i \neq s'_j$
2.  $s_i \neq s_j$  and  $s'_i = s'_j$ .

It immediately follows from the definition that:

**Lemma 4.** *Given two equal-length strings,  $s$  and  $s'$ , if  $(i, j)$  is a mismatch pair between  $s$  and  $s'$ , then for every bijection  $\Pi : \Sigma_s \rightarrow \Sigma_{s'}$  either location  $i$  or location  $j$ , or both locations  $i$  and  $j$  are mismatches, i.e.  $\pi(s_i) \neq s'_i$  or  $\pi(s_j) \neq s'_j$  or both are unequal.*

Conversely,

**Lemma 5.** *Let  $s$  and  $s'$  be two equal-length strings and let  $S \subset \{1, \dots, m\}$  be a set of locations which does not contain any mismatch pair. Then there exists a bijection  $\pi : \Sigma_s \rightarrow \Sigma_{s'}$  that is parameterized on  $S$ , i.e. for every  $i \in S$ ,  $\pi(s_i) = s'_i$ .*

The idea of our algorithm is to count the mismatch pairs between  $s$  and  $s'$ . Since each pair contributes *at least* one error in the p-match between  $s$  and  $s'$  it follows immediately from Lemma 4 that if there are more than  $k$  mismatch pairs then  $s$  does not parameterize  $k$ -match  $s'$ . We claim that if there are fewer than  $k/2 + 1$  mismatch pairs then  $s$  parameterize  $k$ -matches  $s'$ .

**Definition 2.** *Given two equal-length strings,  $s$  and  $s'$ , a collection  $L$  of mismatch pairs is said to be a maximal disjoint collection if (1) all mismatch pairs in  $L$  are disjoint, i.e. do not share a common location, and (2) there is no mismatch pair that can be added to  $L$  without violating disjointness.*

**Corollary 1.** *Let  $s$  and  $s'$  be two strings, and let  $L$  be a maximal disjoint collection of mismatch pairs of  $s$  and  $s'$ . If  $|L| > k$ , then for every  $\Pi : \Sigma_s \rightarrow \Sigma'_{s'}$ , the  $\Pi$ -mismatch is greater than  $k$ . If  $|L| \leq k/2$  then there exists  $\Pi : \Sigma_s \rightarrow \Sigma'_{s'}$  such that the  $\Pi$ -mismatch counter is less than or equal to  $k$ .*

*Proof.* Combining Lemma 4 and Lemma 5 yields the proof. □

### 3.3 String Comparison Algorithm

The method uses Corollary 1. First the mismatch pairs need to be found. In fact, by Corollary 1 only  $k + 1$  of them need to be found since  $k + 1$  mismatch pairs implies that there is no parameterized  $k$ -match. After the mismatch pairs are found if the number of mismatch pairs  $mp$  is less than  $k/2$  or more than  $k$  we can announce a mismatch or match immediately according to Corollary 1. The difficult case to detect is whether there indeed is a parameterized  $k$ -match

for  $k/2 < mp \leq k$ . This is done with the bipartite matching algorithm. However, here the bipartite graph needs to be constructed somewhat differently. While the case where  $mp \leq k/2$  implies an immediate match, for simplicity of presentation we will not differentiate between the case of  $mp \leq k/2$  and  $k/2 < mp \leq k$ . Thus from here on we will only bother with the cases  $mp \leq k$  and  $mp > k$ .

**Find Mismatch Pairs.** In order to find the mismatch pairs of equal-length strings  $s$  and  $s'$  we use a simple stack scheme, one stack for each character in the alphabet. First mismatch pairs are searched for according to the first rule of definition 1, i.e.  $s_i = s_j$  but  $s'_i \neq s'_j$ . Then within the leftover locations mismatch pairs are sought that satisfy the second rule of definition 1, i.e.  $s_i \neq s_j$  but  $s'_i = s'_j$ .

Algorithm: Find mismatch pairs( $s, s'$ )

1. Construct a stack  $S_\sigma$  for each  $\sigma \in \Sigma_s$ .
2. For  $i = 1$  to  $m$  do
  - if  $\text{empty}(S_{s_i})$  then  $\text{push}(S_{s_i}, \langle s'_i, i \rangle)$
  - else if  $\text{top}(S_{s_i}) = \langle s'_i, * \rangle$  then  $\text{push}(S_{s_i}, \langle s'_i, i \rangle)$
  - else  $\langle s'_*, j \rangle \leftarrow \text{pop}(S_{s_i});$  add  $(i, j)$  to the mismatch pairs
3. Set  $\bar{L}$  to be all locations of  $\{1, \dots, m\}$  that do not appear in the mismatch pairs.
4. Reverse the roles of  $s$  and  $s'$  and do step 2 while scanning items only from  $\bar{L}$   
(i.e. here step 2 begins - "For  $j \in \bar{L}$  do")

**Time Complexity:** The algorithm that finds all mismatch pairs between two strings of length  $m$  runs in  $O(m)$  time. Each character in the string is pushed / popped onto at most one stack.

**Verification.** The verification is performed only when the number of mismatch pairs that were found,  $mp$ , satisfies  $mp \leq k$ . Verification consists of a procedure that finds the minimal  $\pi$ -mismatch over all bijections  $\pi$ . The technique used is similar to the bipartite matching algorithm discussed in the beginning of the section.

Let  $\hat{L} \subset \{1, \dots, m\}$  be the locations that appear in a mismatch pair. We say that a symbol  $a \in \Sigma_s$  is *mismatched* if there is a location  $i \in \hat{L}$  such that  $s_i = a$ . Likewise,  $b \in \Sigma_{s'}$  is *mismatched* if there is a location  $i \in \hat{L}$  such that  $s'_i = b$ . A symbol  $a \in \Sigma_s$ , or  $b \in \Sigma_{s'}$ , is *free* if it is not mismatched.

Construct a bipartite graph  $B = (U \cup V, E)$  defined as follows.  $U$  contains all mismatched symbols  $a \in \Sigma_s$  and  $V$  contains all mismatched symbols  $b \in \Sigma_{s'}$ . Moreover,  $U$  contains all free symbols  $a \in \Sigma_s$  for which there is a location

$i \in \{1, \dots, m\} - \hat{L}$  such that  $s_i = a$  and  $s'_i$  is a mismatched symbol. Likewise,  $V$  contains all free symbols  $b \in \Sigma_{s'}$  for which there is a location  $i \in \{1, \dots, m\} - \hat{L}$  such that  $s'_i = b$  and  $s_i$  is a mismatched symbol. The edges, as before, have weights that correspond to the number of locations where they are aligned with each other.

**Lemma 6.** *The bipartite graph  $B$  is of size  $O(k)$ . Moreover, given the mismatch pairs it can be constructed in  $O(k)$  time.*

While the lemma states that the bipartite graph is of size  $O(k)$  it still may be the case that the edge weights may be substantially larger. However, if there exists an edge  $e = (a, b)$  with weight  $> k$  then it must be that  $\pi(a) = b$  for otherwise we immediately have  $> k$  mismatches. Thus, before computing the maximum matching, we drop every edge with weight  $> k$ , along with their vertices. What remains is a bipartite graph with edge weights between 1 and  $k$ .

**Theorem 1.** *Given two equal-length strings  $s$  and  $s'$ , with  $mp \leq k$  mismatch pairs. It is possible to verify whether there is a parameterized  $k$ -match between  $s$  and  $s'$  in  $O(k^{1.5})$  time.*

**Time Complexity:** Given two equal-length strings  $s$  and  $s'$ , it is possible to determine whether  $s$  parameterized  $k$ -matches  $s'$  in  $O(m + k^{1.5})$  time,  $O(m)$  to find the mismatch pairs and  $O(k^{1.5})$  to check these pairs with the appropriate bipartite matching.

## 4 The Algorithm

We are now ready to introduce our algorithm for the problem of parameterized pattern matching with  $k$  mismatches:

- **Input:** Two strings,  $t = t_1, t_2, \dots, t_n$  and  $p = p_1, p_2, \dots, p_m$ , and an integer  $k$ .
- **Output:** All locations  $i$  in  $t$  where  $p$  parameterized  $k$ -matches  $t_i, \dots, t_{i+m-1}$ .

Our algorithm has two phases, the *pattern preprocessing phase* and the *text scanning phase*. In this section we present the text scanning phase and will assume that the pattern preprocessing phase is given. In the following section we describe an efficient method to preprocess the pattern for the needs of the text scanning phase.

**Definition 3.** *Let  $s$  and  $s'$  be two-equal length strings. Let  $L$  be a collection of disjoint mismatch pairs between  $s$  and  $s'$ . If  $L$  is maximal and  $|L| \leq k$  then  $L$  is said to be a  $k$ -good witness for  $s$  and  $s'$ . If  $|L| > k$  then  $L$  is said to be a  $k$ -bad witness for  $s$  and  $s'$ .*

The text scanning phase has two stages the (a) *filter stage* and the (b) *verification stage*. In the filter stage for each text location  $i$  we find either a  $k$ -good witness or a  $k$ -bad witness for  $p$  and  $t_i \dots t_{i+m-1}$ . Obviously, by Corollary 1, if there is a  $k$ -bad witness then  $p$  cannot parameterize  $k$ -match  $t_i \dots t_{i+m-1}$ . Hence, after the filter stage, it remains to verify for those locations  $i$  which have a  $k$ -good witness whether  $p$  parameterize  $k$ -matches  $t_i \dots t_{i+m-1}$ . The verification stage is identical to the verification procedure in Section 3.3 and hence we will not dwell on this stage.

#### 4.1 The Filter Stage

The underlying idea of the filter stage is similar to [16]. One location after another is evaluated utilizing the knowledge accumulated at the previous locations combined with the information gathered in the pattern preprocessing stage. The information of the pattern preprocessing stage is:

**Output of pattern preprocessing:** For each location  $i$  of  $p$ , a maximal disjoint collection of mismatch pairs  $L$  for some pattern prefix  $p_1, \dots, p_{j-i+1}$  and  $p_i, \dots, p_j$  such that either  $|L| = 3k + 3$  or  $j = m$  and  $|L| \leq 3k + 3$ .

As the first step of the algorithm we consider the first text location, i.e. when text  $t_1 \dots t_m$  is aligned with  $p_1 \dots p_m$ . Using the method in Section 3.3 we can find all the mismatch pairs in  $O(m)$  time, and hence find a  $k$ -good or  $k$ -bad witness for the first text location. When evaluating subsequent locations  $i$  we maintain the following invariant: *For all locations  $l < i$  we have either a  $k$ -good or  $k$ -bad witness for location  $l$ .*

It is important to observe that, when evaluating location  $i$  of the text, if we discover a maximal disjoint collection of mismatch pairs of size  $k + 1$ , i.e. a  $k$ -bad witness, between a prefix  $p_1, \dots, p_j$  of the pattern and  $t_i, \dots, t_{i+j-1}$  we will stop our search, since this immediately implies that  $p_1, \dots, p_m$  and  $t_i, \dots, t_{i+m-1}$  have a  $k$ -bad witness. We say that  $i + j$  is a *stop location* for  $i$ . If we have a  $k$ -good witness at location  $i$  then  $i + m$  is its *stop location*. When evaluating location  $i$  of the text, each of the previous locations has a stop location. The location  $\ell$  which has a maximal stop location, over all locations  $l < i$ , is called a *maximal stopper* for  $i$ .

The reason for working with the maximal stopper for  $i$  is that the text beyond the maximal stopper has not yet been scanned. If we can show that the time to compute a  $k$ -bad witness or find a maximal disjoint collection of mismatch pairs for location  $i$  up until the maximal stopper, i.e. for  $p_1, \dots, p_{\ell-i+1}$  with  $t_i, \dots, t_{\ell}$ , is  $O(k)$  then we will spend overall  $O(n + nk)$  time for computing,  $O(nk)$  to compute up until the maximal stopper for each location and  $O(n)$  for the scanning and updating the current maximal collection of mismatch pairs.

**Up to the Maximal Stopper.** The situation now is that we are evaluating a location  $i$  of  $t$ . Let  $\ell$  be the maximal stopper for  $i$ . We utilize two pieces of precomputed information; (1) the pattern preprocessing for location  $i - \ell + 1$  of  $p$  and (2) the  $k$ -good witness or  $k$ -bad witness (of size  $k + 1$ ) for location  $\ell$  of  $t$ . Let  $\ell'$  be the stop location of  $\ell$ . We would like to evaluate the overlap of  $p_1, \dots, p_{\ell' - i + 1}$  with  $t_i, \dots, t_{\ell'}$  and to find a maximal disjoint collection  $L$  of mismatch pairs on this overlap, or, if it exists, a maximal disjoint collection  $L$  of mismatch pairs of size  $k + 1$  on a prefix of the overlap.

There are two cases. One possibility is that the pattern preprocessing returns  $3k + 3$  mismatch pairs for a prefix  $p_1, \dots, p_j$  and  $p_{\ell - i + 1}, \dots, p_{\ell - i + j + 1}$  where  $j \leq \ell' - i + 1$ . Yet, there are  $\leq k + 1$  mismatch pairs between  $p_1, \dots, p_{\ell' - \ell + 1}$  and  $t_\ell, \dots, t_{\ell'}$ .

**Lemma 7.** *Let  $s, s'$  and  $s''$  be three equal-length strings such that there is a maximal collection of mismatch pairs  $L_{s, s'}$  between  $s$  and  $s'$  of size  $\leq M$  and a maximal collection of mismatch pairs  $L_{s', s''}$  between  $s'$  and  $s''$  of size  $\geq 3M$ . Then there must be a maximal collection of mismatch pairs  $L_{s, s''}$  between  $s$  and  $s''$  of size  $\geq M$ .*

Set  $M$  to be  $k + 1$  and one can use the Lemma for the case at hand, namely, there must be at least  $k + 1$  mismatch pairs between  $p_1, \dots, p_{\ell' - i + 1}$  and  $t_i, \dots, t_{\ell'}$ , which defines a  $k$ -bad witness.

The second case is where the pattern preprocessing returns fewer than  $3k + 3$  mismatch pairs or it returns  $3k + 3$  mismatch pairs but it does so for a prefix  $p_1, \dots, p_j$  and  $p_{\ell - i + 1}, \dots, p_{\ell - i + j + 1}$  where  $j > \ell' - i + 1$ . However, we still have an upper bound of  $k + 1$  mismatch pairs between  $p_1, \dots, p_{\ell' - \ell + 1}$  and  $t_\ell, \dots, t_{\ell'}$ . So, we can utilize the fact that:

**Lemma 8.** *Given three strings,  $s, s', s''$  such that there are maximal disjoint collections of mismatch pairs,  $L_{s, s'}$  and  $L_{s', s''}$  of sizes  $O(k)$ . In  $O(k)$  time one can find a  $k$ -good or  $k$ -bad witness for  $s$  and  $s''$ .*

**Putting it Together.** The filter stage takes  $O(nk)$  time and the verification stage takes  $O(nk^{1.5})$ . Hence,

**Theorem 2.** *Given the preprocessing stage we can announce for each location  $i$  of  $t$  whether  $p$  parameterized  $k$ -matches  $t_i, \dots, t_{i+m-1}$  in  $O(nk^{1.5})$  time.*

## 5 Pattern Preprocessing

In this section we solve the pattern preprocessing necessary for the general algorithm.



- **Input:** A pattern  $p = p_1, \dots, p_m$  and an integer  $k$ .
- **Output:** For each location  $i$ ,  $1 \leq i \leq m$  a maximal disjoint collection of mismatch pairs  $L$  for the minimal pattern prefix  $p_1, \dots, p_{j-i+1}$  and  $p_i, \dots, p_j$  such that either  $|L| = 3k + 3$  or  $j = m$  and  $|L| \leq 3k + 3$ .

The naive method takes  $O(m^2)$ , by applying the method from Section 3.3. However, this does not exploit any previous information for a new iteration. Since every alignment combines two previous alignments, we can get a better result. Assume, w.l.o.g., that  $m$  is a power of 3. We divide the set of alignments into  $\log_3 m + 1$  sequences as follows;

$$R_1 = [2, 3], R_3 = [4, 5, 6, 7, 8, 9], \dots, R_i = [3^{i-1} + 1, \dots, 3^i], \dots, R_{\log_3 m} = [3^{\log_3 m - 1} + 1, \dots, 3^{\log_3 m}] \quad 1 \leq i \leq \log_3 m.$$

At each step, we compute the desired mismatches for each set of alignments  $R_i$ . Step  $i$  uses the information computed in steps  $1 \dots i-1$ . We further divide each set into two halves, and compute the first half followed by the second half. This is possible since each  $R_i$  contains an even number of elements, as  $3^i - 3^{i-1} = 2 \cdot 3^{i-1}$ . We split each sequence  $R_i$  into two equal length sequences  $R_i^1[3^{i-1} + 1, \dots, 2 \cdot 3^{i-1}]$  and  $R_i^2 = [2 \cdot 3^{i-1} + 1, \dots, 3^i]$ . For each of the new sequences we have that:

**Lemma 9.** *If  $r_1, r_2 \in R_i^1$  (the first half of set  $R_i$ ) or  $r_1, r_2 \in R_i^2$  (the second half of  $R_i$ ) such that  $r_1 < r_2$  then  $r_2 - r_1 \in R_j$  for  $j < i$ .*

This gives a handle on how to compute our desired output. Denote  $f_i = \min\{j \in R_i^1\}$  and  $m_i = \min\{j \in R_i^2\}$  the representatives of their sequences. We compute the following two stages for each group  $R_i$ , in order  $R_1, \dots, R_{\log_3 m}$ ,

1. Compute a maximal disjoint collection of mismatch pairs  $L$  for some pattern prefix  $p_1, \dots, p_{j+1}$  and  $p_{f_i}, \dots, p_{f_i+j}$  such that  $|L| = (3k+3)3^{\log_3 m - i}$  or  $f_i + j = m$  and  $|L| \leq (3k+3)3^{\log_3 m - i}$ . Do the same with  $p_{m_i}, \dots, p_{m_i+j}$ .
2. Now for each  $j \in R_i^1$  apply the algorithm for the text described in the previous section on the pattern  $p$ , the pattern shifted by  $f_i$  ( $R_i^1$ 's representative) and the pattern shifted by  $j$ . Do the same with  $m_i$  for  $R_i^2$ .

The central idea and importance behind the choice of the number of mismatch pairs that we seek is to satisfy Lemma 7. It can be verified that indeed our choice of sizes always satisfies that there are 3 times as many mismatch pairs as in the previous iteration. Therefore,

**Theorem 3.** *Given a pattern  $p$  of length  $m$ . It is possible to precompute its  $3k+3$  mismatch pairs at each alignment in  $O(km \log_3 m)$  time.*

**Corollary 2.** *Given a pattern  $p$  and text  $t$ , we can solve the parameterized matching with  $k$  mismatches problem in  $O(nk^{1.5} + km \log m)$  time.*

## References

1. K. W. Church A. Amir and E. Dar. Separable attributes: a technique for solving the submatrices character count problem. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 400–401, 2002.
2. A. Amir, Y. Aumann, R. Cole, M. Lewenstein, and E. Porat. Function matching: Algorithms, applications and a lower bound. In *Proc. of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 929–942, 2003.
3. A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM J. Comp.*, 23(2):313–323, 1994.
4. A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49:111–115, 1994.
5. A. Apostolico, P. Erdős, and M. Lewenstein. Parameterized matching with mismatches. *manuscript*.
6. B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proc. 25th Annual ACM Symposium on the Theory of Computation*, pages 71–80, 1993.
7. B. S. Baker. Parameterized string pattern matching. *J. Comput. Systems Sci.*, 52, 1996.
8. B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. on Computing*, 26, 1997.
9. B. S. Baker. Parameterized diff. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 854–855, 1999.
10. R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. In *Proc. 32nd ACM STOC*, pages 407–415, 2000.
11. M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their use in improved network optimizations algorithms. *J. of the ACM*, 34(3):596–615, 1987.
12. H. N. Gabow. Scaling algorithms for network problems. *J. of Computer and System Sciences*, 31:148–168, 1985.
13. H. N. Gabow and R.E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. on Computing*, 18:1013–1036, 1989.
14. Z. Galil and R. Giancarlo. Improved string matching with  $k$  mismatches. *SIGACT News*, 17(4):52–54, 1986.
15. S. R. Kosaraju. Faster algorithms for the construction of parameterized suffix trees. *Proc. 36th IEEE FOCS*, pages 631–637, 1995.
16. G.M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching. *Proc. 18th ACM Symposium on Theory of Computing*, pages 220–230, 1986.
17. G.M. Landau and U. Vishkin. Fast string matching with  $k$  differences. *J. Comput. Syst. Sci.*, 37, 1988.
18. V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.
19. B.M. Mehtre G.P. Babu and M.S. Kankanhalli. Color indexing for efficient image retrieval. *Multimedia Tools and Applications*, 1(4), 1995.
20. W.K. Sung M.Y. Kao, T.W. Lam and H.F. Ting. A decomposition theorem for maximum weight bipartite matching. *SIAM J. on Computing*, 31(1):18–26, 2001.
21. M. Swain and D. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.

# Approximation of Rectangle Stabbing and Interval Stabbing Problems

Sofia Kovaleva<sup>1</sup> and Frits C.R. Spieksma<sup>2</sup>

<sup>1</sup> Department of Quantitative Economics, Maastricht University, P.O. Box 616,  
NL-6200 MD Maastricht, The Netherlands,  
`sonja.kovaleva@mail.com`,

<sup>2</sup> Department of Applied Economics, Katholieke Universiteit Leuven, Naamsestraat  
69, B-3000, Leuven, Belgium,  
`frits.spieksma@econ.kuleuven.ac.be`

## 1 Introduction

The weighted rectangle multi-stabbing problem (WRMS) can be described as follows: given is a grid in  $\mathbb{R}^2$  consisting of columns and rows each having a positive integral weight, and a set of closed axis-parallel rectangles each having a positive integral demand. The rectangles are placed arbitrarily in the grid with the only assumption that each rectangle is intersected by at least one column and at least one row. The objective is to find a minimum weight (multi)set of columns and rows of the grid so that for each rectangle the total multiplicity of selected columns and rows stabbing it is at least its demand. (A column or row is said to stab a rectangle if it intersects it.)

We present approximation algorithms for two variants of WRMS (see e.g. Vazirani [6] for an overview on approximation algorithms). First, we describe a  $\frac{q+1}{q}$ -approximation algorithm called *ROUND* for the case where the demand of each rectangle is bounded from below by an integer  $q$ . Observe that this provides a 2-approximation algorithm for the WRMS described above, where  $q = 1$ . Second, we present a deterministic  $\frac{e}{e-1} \approx 1.582$ -approximation algorithm called *STAB* for the case when each rectangle in the input is intersected by exactly one row (this special case is referred to as *WIS*). Our algorithms are based on rounding the linear programming relaxation of an integer programming formulation in an interesting way. We use the following property present in our formulation: the variables can be partitioned into two sets such that when given the values of one set of variables, one can compute in polynomial time the optimal values of the variables of the other set of variables, and vice versa. Next, we consider different ways of rounding one set of variables, and compute each time the values of the remaining variables, while keeping the best solution.

**Motivation.** Although at first sight WRMS may seem rather specific, it comprises a quite natural combinatorial optimization problem: given a set of arbitrary connected figures in  $\mathbb{R}^2$ , stab the figures with the minimum number of straight lines of at most two different directions. Indeed, suppose we are given

$n$  connected point sets (the figures) in  $\mathbb{R}^2$ :  $F^1, F^2, \dots, F^n$ , and two directions, specified by vectors  $v_1$  and  $v_2$ , orthogonal to them. Denote:  $a^i = \min\{\langle v_1, v \rangle \mid v \in F^i\}$ ,  $b^i = \max\{\langle v_1, v \rangle \mid v \in F^i\}$ ,  $c^i = \min\{\langle v_2, v \rangle \mid v \in F^i\}$ ,  $d^i = \max\{\langle v_2, v \rangle \mid v \in F^i\}$ ,  $\forall i = [1 : n]$ . Now the problem can be transformed into the problem of stabbing  $n$  closed axis-parallel rectangles  $\{(x, y) \in \mathbb{R}^2 \mid a^i \leq x \leq b^i, c^i \leq y \leq d^i\}$ ,  $\forall i = [1 : n]$  with a minimum number of axis-parallel lines. By defining a grid whose rows and columns are the horizontal and vertical lines going through the edges of the rectangles, and by observing that any vertical (horizontal) line intersecting a set of rectangles can be replaced by a vertical (horizontal) line intersecting this same set of rectangles while going through an edge of some rectangle, it follows that stabbing arbitrary connected figures in the plane using straight lines of at most two directions is a special case of WRMS.

Notice also that integer linear programming problems of the following type can be reformulated as instances of WRMS:  $\text{minimize}\{wx \mid (B|C)x \geq b, x \in \mathbb{Z}^l\}$ , where  $b \in \mathbb{Z}_+^n$ ,  $w \in \mathbb{Z}_+^l$ , and  $B$  and  $C$  are both 0,1-matrices with consecutive '1'-s in the rows (a so-called interval matrix). Indeed, construct a grid which has a column for each column in  $B$  and a row for each column in  $C$ . For each row  $i$  of matrix  $B|C$ , draw a rectangle  $i$  such that it intersects only the columns and rows of the grid corresponding to the positions of '1'-s in row  $i$ . Observe that this construction is possible since  $B$  and  $C$  have consecutive '1'-s in the rows. To complete the construction, assign demand  $b_i$  to each rectangle  $i$  and a corresponding weight  $w_j$  to each column and row of the grid. Let the decision variables  $x$  describe the multiplicities of the columns and rows of the grid. In this way we have obtained an instance of WRMS. In other words, integer programming problems where the columns of the constraint matrix  $A$  can be permuted such that  $A = (B|C)$  with  $B$  and  $C$  each being an interval matrix, is a special case of WRMS.

**Literature.** WRMS has received attention in literature before. Gaur et al. [2] present a 2-approximation algorithm for WRMS with unit weights and demands, which admits a generalization to arbitrary weights and demands preserving the same approximation factor. Thus, our algorithm *ROUND* with the ratio of  $\frac{q+1}{q}$  is an improvement for those instances where a lower bound on the rectangles' demands  $q$  is larger than 1. Furthermore, Hassin and Megiddo [3] study a number of special cases of the problem of stabbing geometric figures in  $\mathbb{R}^2$  by a minimum number of straight lines. In particular, they present a 2-approximation algorithm for the task of stabbing connected figures of the same shape and size with horizontal and vertical lines. Moreover, they study the case which in our setting corresponds to WIS with unit weights and demands, where each rectangle in the input is intersected by exactly  $K$  columns, and give a  $2 - \frac{1}{K}$ -approximation algorithm for this problem. Our  $e/e - 1$ -approximation algorithm provides a better approximation ratio for  $K \geq 3$  and does not impose any restrictions on the number of columns intersecting rectangles. Finally, the special case of WIS with unit weights and demands, where each rectangle is stabbed by at most two columns, is shown to be APX-hard in [5].

**Our results.** Summarizing our results:

- applying a similar, slightly refined approach, we generalize the results of Gaur et al. [2] to obtain a  $\frac{q+1}{q}$ -approximation algorithm called *ROUND* for the case where the demand of each rectangle is bounded from below by an integer  $q$  (Section 3),
- we describe an  $\frac{e}{e-1}$ -approximation algorithm called *STAB* for WIS based on an original rounding idea (Section 4),
- we state here without proof that the special case of WIS with unit weights where each rectangle is stabbed by at most  $k$  columns admits an  $\frac{1}{1-(1/k)^k}$ -approximation algorithm. For the (difficult) case  $k = 2$ , this implies a  $\frac{4}{3}$ -approximation algorithm.

For some of the proofs of the results in the next sections we refer to Kovaleva [4].

**Applications.** For the sake of conciseness we simply refer to Gaur et al. [2] for an application of WRMS related to image processing and numerical computations, and to Hassin and Megiddo [3] for military and medical applications.

## 2 Preliminaries

Let us formalize the definition of WRMS. Let the grid in the input consist of  $t$  columns and  $m$  rows, numbered consecutively from left to right and from bottom to top, with positive weight  $w_c$  ( $v_r$ ) attached to each column  $c$  (row  $r$ ). Further, we are given  $n$  rectangles such that rectangle  $i$  has demand  $d_i \in \mathbb{Z}_+$  and is specified by leftmost column  $l_i$ , rightmost column  $r_i$ , top row  $t_i$  and bottom row  $b_i$ .

Let us give a natural ILP formulation of WRMS. In this paper we use notation  $[a : b]$  for the set of integers  $\{a, a + 1, \dots, b\}$ . The decision variables  $y_c, z_r \in \mathbb{Z}_+$ ,  $c \in [1 : t], r \in [1 : m]$ , denote the multiplicities of column  $c$  and row  $r$  respectively.

$$\text{Minimize} \quad \sum_{c=1}^t w_c y_c + \sum_{r=1}^m v_r z_r \quad (1)$$

$$\text{subject to} \quad \sum_{r \in [b_i : t_i]} z_r + \sum_{c \in [l_i : r_i]} y_c \geq d_i \quad \forall i \in [1 : n] \quad (2)$$

$$z_r, y_c \in \mathbb{Z}_+^1 \quad \forall r, c. \quad (3)$$

The linear programming relaxation is obtained when replacing the integrality constraints (3) by the nonnegativity constraints  $z_r, y_c \geq 0$ ,  $\forall r, c$ .

For an instance  $\mathcal{I}$  of WRMS and a vector  $b \in \mathbb{Z}^n$ , we introduce two auxiliary ILP problems:

$$\begin{aligned} \text{IP}^y(\mathcal{I}, b): \quad & \text{Minimize} \quad \sum_{c=1}^t w_c y_c \\ & \text{subject to} \quad \sum_{c \in [l_i : r_i]} y_c \geq b_i \quad \forall i \in [1 : n] \\ & \quad \quad y_c \in \mathbb{Z}_+, \quad \forall c \in [1 : t] \end{aligned} \quad (4)$$

$$\begin{aligned} \text{IP}^z(\mathcal{I}, b): \quad & \text{Minimize} \quad \sum_{r=1}^m v_r z_r \\ & \text{subject to} \quad \sum_{r \in [b_i : t_i]} z_r \geq b_i \quad \forall i \in [1 : n] \\ & \quad \quad \quad z_r \in \mathbb{Z}_+, \quad \forall c \in [1 : m] \end{aligned} \quad (5)$$

**Lemma 1.** *For any  $b \in \mathbb{Z}^n$ , the LP-relaxation of each of the problems  $\text{IP}^z(\mathcal{I}, b)$  and  $\text{IP}^y(\mathcal{I}, b)$  is integral.*

**Proof.** This follows from the unimodularity of the constraint matrix of (5) which is implied by the “consecutive one’s”-property (see e.g. Ahuja et al. [1]).  $\square$

**Corollary 2.** *The optimum value of  $\text{IP}^y(\mathcal{I}, b)$  ( $\text{IP}^z(\mathcal{I}, b)$ ) is smaller than or equal to the value of any feasible solution to its LP-relaxation.*

**Corollary 3.** *The problem  $\text{IP}^y(\mathcal{I}, b)$  ( $\text{IP}^z(\mathcal{I}, b)$ ) can be solved in polynomial time. Its optimal solution coincides with that of its LP-relaxation.*

In fact, the special structure of  $\text{IP}^y(\mathcal{I}, b)$  ( $\text{IP}^z(\mathcal{I}, b)$ ) allows to solve it via a minimum-cost flow algorithm.

**Lemma 4.** *The problem  $\text{IP}^y(\mathcal{I}, b)$  ( $\text{IP}^z(\mathcal{I}, b)$ ) can be solved in time  $O(\text{MCF}(t, n + t))$  ( $O(\text{MCF}(m, n + m))$ ), where  $\text{MCF}(p, q)$  is the time needed to solve the minimum cost flow problem on a network with  $p$  nodes and  $q$  arcs.*

**Proof.** Consider the LP-relaxation of formulation  $\text{IP}^y(\mathcal{I}, b)$  and substitute variables with new variables  $u_0, \dots, u_t$  as  $y_c = u_c - u_{c-1}$ ,  $\forall c \in [1 : t]$ . Then it transforms into

$$\begin{aligned} \text{Minimize} \quad & -w_1 u_0 + (w_1 - w_2) u_2 + \dots + (w_{t-1} - w_t) u_{t-1} + w_t u_t \\ \text{subject to} \quad & u_{r_i} - u_{l_i-1} \geq b_i, \quad \forall i \in [1 : n] \\ & u_c - u_{c-1} \geq 0, \quad \forall c \in [1 : t]. \end{aligned} \quad (6)$$

Let us denote the vector of objective coefficients, the vector of right-hand sides and the constraint matrix by  $w$ ,  $b$  and  $C$  respectively, and the vector of variables by  $u$ . Then (4) can be represented as  $\{\text{minimize } wu \mid Cu \geq b\}$ . Its dual is:  $\{\text{maximize } bx \mid C^T x = w, x \geq 0\}$ . Observe that this is a minimum cost flow formulation with flow conservation constraints  $C^T x = w$ , since  $C^T$  has exactly one ‘1’ and one ‘-1’ in each column. Given an optimal solution to the minimum cost flow problem, one can easily obtain the optimal dual solution  $u_0, \dots, u_t$  (see Ahuja et al. [1]), and thus optimal  $y_1, \dots, y_t$  as well.  $\square$

### 3 A $\frac{q+1}{q}$ —Approximation Algorithm for WRMSq

Let WRMSq be a special case of WRMS, where  $d_i \geq q$ ,  $\forall i \in [1 : n]$ . In this section we describe an algorithm *ROUND*. Given an instance of WRMSq, the algorithm works as follows:

1. solve the LP-relaxation of (1)-(3) for  $\mathcal{I}$  and obtain its optimal solution  $(y^{\text{lp}}, z^{\text{lp}})$
2.  $a_i \leftarrow \lfloor \frac{q+1}{q} \sum_{c \in [l_i:r_i]} y_c^{\text{lp}} \rfloor$ , for all  $i \in [1:n]$ ;
3. solve  $\text{IP}^y(\mathcal{I}, a)$  and obtain  $\bar{y}$ ;
4.  $b_i \leftarrow \lfloor \frac{q+1}{q} \sum_{c \in [b_i:t_i]} z_c^{\text{lp}} \rfloor$ , for all  $i \in [1:n]$ ;
5. solve  $\text{IP}^z(\mathcal{I}, b)$  and obtain  $\bar{z}$ ;
6. return  $(\bar{y}, \bar{z})$

**Fig. 1.** Algorithm *ROUND*.

**Theorem 1.** *Algorithm ROUND is a  $\frac{q+1}{q}$ -approximation algorithm for the problem WRMSq.*

**Proof.** Let  $\mathcal{I}$  be an instance of WRMSq. First, we show that the solution  $(\bar{y}, \bar{z})$  returned by *ROUND* is feasible for  $\mathcal{I}$ , i.e., satisfies constraints (2) and (3). Obviously, vectors  $\bar{y}$  and  $\bar{z}$  are integral, hence, constraint (3) is satisfied. For each  $i \in [1:n]$ , consider the left-hand side of constraint (2) for  $(\bar{y}, \bar{z})$ . By construction,  $\bar{y}$  and  $\bar{z}$  are feasible to  $\text{IP}^y(\mathcal{I}, a)$  and  $\text{IP}^z(\mathcal{I}, b)$  respectively. Using this, and by construction of the vectors  $a$  and  $b$ :

$$\sum_{r \in [b_i:t_i]} \bar{z}_r + \sum_{c \in [l_i:r_i]} \bar{y}_c \geq a_i + b_i = \lfloor \frac{q+1}{q} \sum_{c \in [l_i:r_i]} y_c^{\text{lp}} \rfloor + \lfloor \frac{q+1}{q} \sum_{r \in [b_i:t_i]} z_r^{\text{lp}} \rfloor. \quad (7)$$

Since  $\lfloor \alpha \rfloor + \lfloor \beta \rfloor \geq \lfloor \alpha + \beta \rfloor - 1$ , for any positive  $\alpha$  and  $\beta$ , and since  $z^{\text{lp}}$  and  $y^{\text{lp}}$  satisfy constraint (2), the right-hand side of (7) is greater or equal to:

$$\lfloor \frac{q+1}{q} (\sum_{c \in [l_i:r_i]} y_c^{\text{lp}} + \sum_{r \in [b_i:t_i]} z_r^{\text{lp}}) \rfloor - 1 \geq \lfloor \frac{q+1}{q} d_i \rfloor - 1 \geq d_i + 1 - 1 = d_i,$$

where the last inequality holds since we have  $d_i \geq q$ ,  $\forall i \in [1:n]$  in  $\mathcal{I}$ . Thus, inequality (2) holds for  $(\bar{y}, \bar{z})$  and hence  $(\bar{y}, \bar{z})$  constitute a feasible solution to the instance  $\mathcal{I}$  of WRMSq.

The approximation ratio of *ROUND* is obtained from the ratio between the value of the output solution  $(\bar{y}, \bar{z})$  and the value of the optimal fractional solution  $(y^{\text{lp}}, z^{\text{lp}})$ , which is at most the optimum value of WRMSq.

We claim that  $\sum_{c=1}^t w_c \bar{y}_c \leq \frac{q+1}{q} \sum_{c=1}^t w_c y_c^{\text{lp}}$ . Observe that  $\frac{q+1}{q} y^{\text{lp}}$  is a feasible solution to the LP-relaxation of  $\text{IP}^y(\mathcal{I}, a)$  with  $a_i = \lfloor \frac{q+1}{q} \sum_{c \in [l_i:r_i]} y_c^{\text{lp}} \rfloor$ ,  $\forall i \in [1:n]$ . Indeed, the constraints of (4) are clearly satisfied:  $\sum_{c \in [l_i:r_i]} \frac{q+1}{q} y_c^{\text{lp}} \geq \lfloor \frac{q+1}{q} \sum_{c \in [l_i:r_i]} y_c^{\text{lp}} \rfloor$ ,  $\forall i \in [1:n]$ . Thus, vectors  $\bar{y}$  and  $\frac{q+1}{q} y^{\text{lp}}$  are respectively an optimal solution to  $\text{IP}^y(\mathcal{I}, b^1)$  and a feasible solution to its LP relaxation. Now Corollary 2 implies the claim.

Similarly,  $\sum_{r=1}^m v_r \bar{z}_r \leq \frac{q+1}{q} \sum_{r=1}^m v_r z_r^{\text{lp}}$ . This proves the ratio of  $\frac{q+1}{q}$  between the value of solution  $\bar{y}, \bar{z}$  and solution  $y^{\text{lp}}, z^{\text{lp}}$ .

The running time of the algorithm is determined by the time needed to solve the LP-relaxation of (1)-(3) ( $LP(n, t + m)$ ) and the problems  $\text{IP}^y(\mathcal{I}, a)$  and

$\text{IP}^z(\mathcal{I}, b)$ , which can be solved in time  $MCF(m, m+n)$  and  $MCF(t, t+n)$  (see Lemma 4).  $\square$

**Remark 1.** *There exist instances of WRMS<sub>q</sub> where the ratio between the optimum value of the formulation (1)-(3) and the optimum value of its LP-relaxation is arbitrarily close to  $\frac{q+1}{q}$  (see Kovaleva [4]). This suggests that it is unlikely to improve the approximation ratio of ROUND by another LP-rounding algorithm using the same formulation of WRMS<sub>q</sub>.*

## 4 An $e/(e-1)$ -Approximation Algorithm for WIS

The weighted interval stabbing problem (WIS) is a special case of WRMS with unit rectangles' demands ( $d_i = 1, \forall i \in [1 : n]$ ) and each rectangle in the input intersecting exactly one row of the grid. Hence, we can think of the rectangles as of line segments, or intervals, lying on the rows of the grid. Let interval  $i$  ( $\forall i \in [1 : n]$ ) be specified by the indices of the leftmost and the rightmost columns intersecting it,  $l_i$  and  $r_i$  respectively, and the index of its row  $\rho_i$ .

In this section we describe an algorithm *STAB* for WIS and show that it is a  $e/(e-1)$ -approximation algorithm. Let us first adapt the ILP formulation (1)-(3) to WIS:

$$\text{Minimize } \sum_{c=1}^t w_c y_c + \sum_{r=1}^m v_r z_r \quad (8)$$

$$\text{subject to } z_{\rho_i} + \sum_{c \in (l_i : r_i)} y_c \geq 1 \quad \forall i \in [1 : n] \quad (9)$$

$$z_r, y_c \in \mathbb{Z}_+ \quad \forall r, c. \quad (10)$$

Informally, algorithm *STAB* can be described as follows: given an instance  $\mathcal{I}$  of WIS, the algorithm solves the LP-relaxation of (8)-(10), and obtains an optimal fractional solution  $(y^{\text{lp}}, z^{\text{lp}})$ . Then it constructs  $m+1$  candidate solutions and returns the best of them. Candidate solution number  $j$  ( $j = 0, \dots, m$ ) is constructed as follows: set the largest  $j$   $z$ -values in the fractional solution to 1, the rest to 0; this gives an integral vector  $\bar{z}$ . Solve the problem  $\text{IP}^y(\mathcal{I}, b)$  (5), where  $b_i = 1 - \bar{z}_{\rho_i}$ ,  $\forall i \in [1 : n]$ , and obtain integral  $\bar{y}$ . Now  $(\bar{y}, \bar{z})$  is the  $j$ th candidate solution. A formal description of *STAB* is shown in Figure 2, where we use the following definition:  $\text{value}(y, z) \equiv \sum_{c=1}^t w_c y_c + \sum_{r=1}^m v_r z_r$ .

In order to formulate and prove the main result of this section, we first formulate the following lemma.

**Lemma 5.** *Suppose we are given numbers  $1 > \Delta_1 \geq \Delta_2 \geq \dots \geq \Delta_m \geq 0$ ,  $\forall i = 1, \dots, m$ , and  $\Delta_{m+1} = 0$ . Further, given are positive numbers  $a_1, a_2, \dots, a_m$  and  $Y$ . Then we have:*

$$\min_{j=0, \dots, m} \left( \sum_{r=1}^j a_r + \frac{1}{1 - \Delta_{j+1}} Y \right) \leq \frac{e}{e-1} \left( \sum_{r=1}^m a_r \Delta_r + Y \right). \quad (11)$$

We give the proof of this lemma at the end of this section.



1. solve the LP-relaxation of (8)-(10), and obtain its optimal solution  $(z^{\text{lp}}, y^{\text{lp}})$ ;
2. reindex the rows of the grid so that  $z_1^{\text{lp}} \geq z_2^{\text{lp}} \geq \dots \geq z_m^{\text{lp}}$ ;
3.  $V \leftarrow \infty$ ;
4. for  $j = 0$  to  $m$ 
  - for  $i = 1$  to  $j$   $\bar{z}_i \leftarrow 1$ ,
  - for  $i = j + 1$  to  $m$   $\bar{z}_i \leftarrow 0$ .
  - solve  $\text{IP}^y(\mathcal{I}, b)$ , where  $b_i = 1 - \bar{z}_{\rho_i}$ ,  $\forall i \in [1 : n]$ , and obtain  $\bar{y}$ ;
  - if  $\text{value}(\bar{y}, \bar{z}) < V$  then  $V \leftarrow \text{value}(\bar{y}, \bar{z})$ ,  $y^* \leftarrow \bar{y}$ ,  $z^* \leftarrow \bar{z}$ ;
5. return  $(y^*, z^*)$ .

**Fig. 2.** Algorithm *STAB*

**Theorem 6.** *Algorithm STAB is a  $e/(e-1) \approx 1.582$ -approximation algorithm for WIS.*

**Proof.** Obviously, algorithm *STAB* is a polynomial-time algorithm (see Corollary 3). It is also easy to verify that it always returns a feasible solution. Let us establish the approximation ratio.

Consider an instance  $\mathcal{I}$  of WIS, and let  $(y^{\text{lp}}, z^{\text{lp}})$  and  $(y^A, z^A)$  be respectively an optimal solution to the LP-relaxation of (8)-(10), and the solution returned by the algorithm for  $\mathcal{I}$ . We show that their values are related as follows:

$$\text{value}(y^A, z^A) \leq \frac{e}{e-1} \text{value}(y^{\text{lp}}, z^{\text{lp}}). \quad (12)$$

Since  $\text{value}(y^{\text{lp}}, z^{\text{lp}})$  is at most the optimal value of WIS, the theorem follows.

Assume, without loss of generality, that the rows of the grid are sorted so that:  $z_1^{\text{lp}} \geq z_2^{\text{lp}} \geq \dots \geq z_m^{\text{lp}}$ . Further, suppose there are  $p$   $z^{\text{lp}}$ -values equal to 1, i.e.,  $z_1^{\text{lp}} = \dots = z_p^{\text{lp}} = 1$ ,  $1 > z_{p+1}^{\text{lp}} \geq z_{p+2}^{\text{lp}} \geq \dots \geq z_m^{\text{lp}} \geq 0$ .

Let  $(\bar{y}^j, \bar{z}^j)$  be candidate solution number  $j$  constructed by *STAB* for  $\mathcal{I}$ ,  $\forall j \in [0 : m]$ . From the design of *STAB* we know that:

$$\text{value}(y^A, z^A) = \min_{j \in [0 : m]} \text{value}(\bar{y}^j, \bar{z}^j) \leq \min_{j \in [p : m]} \text{value}(\bar{y}^j, \bar{z}^j). \quad (13)$$

Claim 1.  $\text{value}(\bar{y}^j, \bar{z}^j) \equiv w\bar{y}^j + v\bar{z}^j \leq \sum_{r=1}^j v_r + \frac{wy^{\text{lp}}}{1 - z_{j+1}^{\text{lp}}}$ , for any  $j \in [p : m]$ .

Let us prove it. Consider  $(\bar{y}^j, \bar{z}^j)$  for some  $j \in [p : m]$ . By construction:

- $\bar{z}_r^j = 1$ ,  $\forall r \leq j$ ,
- $\bar{z}_r^j = 0$ ,  $\forall r \geq j + 1$ ,
- $\bar{y}^j$  is an optimal solution to  $\text{IP}^y(\mathcal{I}, b)$  with  $b_i = 1 - \bar{z}_{\rho_i}$ ,  $\forall i \in [1 : n]$ .

Clearly:  $v\bar{z}^j \equiv \sum_{r=1}^m v_r \bar{z}_r^j = \sum_{r=1}^j v_r$ . Let us show that

$$w\bar{y}^j \leq \frac{wy^{\text{lp}}}{1 - z_{j+1}^{\text{lp}}}. \quad (14)$$

To prove this, we establish that the fractional solution

$$\frac{1}{1 - z_{j+1}^{\text{lp}}} y^{\text{lp}}, \quad (15)$$

where we set  $z_{m+1}^{\text{lp}} = 0$ , is feasible to the LP relaxation of  $\text{IP}^y(\mathcal{I}, b)$ . Since  $\bar{y}^j$  is optimal to  $\text{IP}^y(\mathcal{I}, b)$ , Corollary 2 implies (14). So, let us prove:

Claim 1.1. Solution (15) is feasible to  $\text{IP}^y(\mathcal{I}, b)$  with  $b_i = 1 - \bar{z}_{\rho_i}^j$ ,  $\forall i \in [1 : n]$ . We show that constraint (4) is satisfied:

$$\frac{1}{1 - z_{j+1}^{\text{lp}}} \sum_{c \in [l_i, r_i]} y_c^{\text{lp}} \geq 1 - \bar{z}_{\rho_i}^j, \quad \text{for any } i \in [1 : n]. \quad (16)$$

Indeed, in case  $\bar{z}_{\rho_i}^j = 1$ , the inequality trivially holds. Otherwise, if  $\bar{z}_{\rho_i}^j = 0$ , it follows from the construction of  $\bar{z}^j$  that  $\rho_i \geq j + 1$ . The ordering of the  $z^{\text{lp}}$ -values implies that  $z_{\rho_i}^{\text{lp}} \leq z_{j+1}^{\text{lp}}$ . Then, using this and the fact that solution  $(y^{\text{lp}}, z^{\text{lp}})$  satisfies constraint (9), we have:

$$\frac{1}{1 - z_{j+1}^{\text{lp}}} \sum_{c \in [l_i, r_i]} y_c^{\text{lp}} \geq \frac{1}{1 - z_{j+1}^{\text{lp}}} (1 - z_{\rho_i}^{\text{lp}}) \geq \frac{1}{1 - z_{j+1}^{\text{lp}}} (1 - z_{j+1}^{\text{lp}}) \geq 1.$$

This proves (16), and subsequently Claim 1.1 and Claim 1.

From (13) and Claim 1:

$$\begin{aligned} \text{value}(y^A, z^A) &\leq \min_{j \in [p:m]} \left( \sum_{r=1}^j v_r + \frac{wy^{\text{lp}}}{1 - z_{j+1}^{\text{lp}}} \right) = \\ &= \sum_{r=1}^p v_r + \min_{j \in [p:m]} \left( \sum_{r=p+1}^j v_r + \frac{wy^{\text{lp}}}{1 - z_{j+1}^{\text{lp}}} \right). \end{aligned}$$

Lemma 5 now implies that the last expression can be upper bounded by

$$\sum_{r=1}^p v_r + \frac{e}{e-1} \left( \sum_{r=p+1}^m v_r z_r^{\text{lp}} + wy^{\text{lp}} \right) \leq \frac{e}{e-1} \left( \sum_{r=1}^p v_r + \sum_{r=p+1}^m v_r z_r^{\text{lp}} + wy^{\text{lp}} \right).$$

Since  $z_1^{\text{lp}} = \dots = z_p^{\text{lp}} = 1$ , the last expression can be rewritten as:

$$\frac{e}{e-1} \left( \sum_{r=1}^m v_r z_r^{\text{lp}} + wy^{\text{lp}} \right) = \frac{e}{e-1} (v z^{\text{lp}} + wy^{\text{lp}}),$$

which establishes inequality (12) and proves the theorem.  $\square$

**Remark 2.** *There exist instances of WIS for which the ratio between the optimum values of (8)-(10) and its LP relaxation is arbitrarily close to  $e/(e-1)$  (see Kovaleva [4]). Thus, it is unlikely to improve the approximation ratio of STAB by another LP-rounding algorithm using the same formulation of WIS.*

Now we give a proof of Lemma 5. This version of the proof is due to Jiří Sgall (see acknowledgements).

**Lemma 5.** *Suppose we are given numbers  $1 > \Delta_1 \geq \Delta_2 \geq \dots \geq \Delta_m \geq 0$  and  $\Delta_{m+1} = 0$ . Further, given are positive numbers  $a_1, a_2, \dots, a_m$  and  $Y$ . Then we have:*

$$\min_{j=0, \dots, m} \left( \sum_{r=1}^j a_r + \frac{Y}{1 - \Delta_{j+1}} \right) \leq \frac{e}{e-1} \left( \sum_{r=1}^m a_r \Delta_r + Y \right). \quad (17)$$

**Proof.** We use mathematical induction on the size of inequality  $m$ . For  $m = 0$ , the statement trivially holds. Suppose, the lemma was proven for any inequality of size smaller than  $m$ . First, consider the case  $\Delta_1 \geq \frac{e-1}{e}$ . We can write:

$$\begin{aligned} \min_{j=0, \dots, m} \left( \sum_{r=1}^j a_r + \frac{Y}{1 - \Delta_{j+1}} \right) &\leq \min_{j=1, \dots, m} \left( \sum_{r=1}^j a_r + \frac{Y}{1 - \Delta_{j+1}} \right) = \\ &= a_1 + \min_{j=1, \dots, m} \left( \sum_{r=2}^j a_r + \frac{Y}{1 - \Delta_{j+1}} \right). \end{aligned}$$

The last minimum is the LHS of (17) for a smaller sequence:  $\Delta_2, \dots, \Delta_m$ , and  $a_2, \dots, a_m$ . Applying the induction hypothesis, we can bound the last expression from above as follows: (we also use our bound on  $\Delta_1$ )

$$\begin{aligned} a_1 + \frac{e}{e-1} \left( \sum_{r=2}^m a_r \Delta_r + Y \right) &\leq a_1 * \Delta_1 \frac{e}{e-1} + \frac{e}{e-1} \left( \sum_{r=2}^m a_r \Delta_r + Y \right) = \\ &= \frac{e}{e-1} \left( \sum_{r=1}^m a_r \Delta_r + Y \right). \end{aligned}$$

So, we have shown an induction step for the case  $\Delta_1 \geq \frac{e-1}{e}$ . For the remaining case,  $\Delta_1 < \frac{e-1}{e}$ , we give a direct proof below.

Suppose  $\Delta_1 < \frac{e-1}{e}$ . Denote the LHS of (17) by  $X$ , and notice that:

$$\sum_{r=1}^j a_r \geq X - \frac{1}{1 - \Delta_{j+1}} Y, \quad \text{for } 0 \leq j \leq m. \quad (18)$$

The following steps are justified below:

$$\begin{aligned} \sum_{r=1}^m a_r \Delta_r + Y &= \sum_{j=1}^m \left( (\Delta_j - \Delta_{j+1}) \sum_{r=1}^j a_r \right) + Y \geq^{(1)} \\ &\geq \sum_{j=1}^m (\Delta_j - \Delta_{j+1}) X - \left( \sum_{j=1}^m \frac{\Delta_j - \Delta_{j+1}}{1 - \Delta_{j+1}} \right) Y + Y = \end{aligned}$$

$$\begin{aligned}
 &= \Delta_1 X - \left( \sum_{j=1}^m \left( \frac{\Delta_j - 1}{1 - \Delta_{j+1}} + 1 \right) \right) Y + Y = \\
 &= \Delta_1 X + \left( 1 - m + \sum_{j=1}^m \frac{1 - \Delta_j}{1 - \Delta_{j+1}} \right) Y \stackrel{(2)}{\geq} \\
 &\geq \Delta_1 X + \left( 1 - m + m(1 - \Delta_1)^{\frac{1}{m}} \right) Y \stackrel{(3)}{\geq} \\
 &\geq \Delta_1 X + \left( 1 - m + m(1 - \Delta_1)^{\frac{1}{m}} \right) (1 - \Delta_1) X = \\
 &= \left( 1 + m(-1 + (1 - \Delta_1)^{\frac{1}{m}})(1 - \Delta_1) \right) X \stackrel{(4)}{\geq} \left( 1 - \frac{1}{e} \right) X.
 \end{aligned}$$

(1) Here we use the ordering of delta's and inequality (18).

(2) This inequality follows from the arithmetic-geometric mean inequality  $\sum_{j=1}^m x_j \geq m(\prod_{j=1}^m x_j)^{1/m}$  used for positive numbers  $x_j = \frac{1 - \Delta_j}{1 - \Delta_{j+1}}$ .

(3) Here we use inequality  $Y \geq (1 - \Delta_1)X$ , which is implied by (18) for  $j = 0$ , and the fact that the coefficient of  $Y$  is non-negative, which follows from  $1 - \Delta_1 \geq \frac{1}{e} \geq (1 - \frac{1}{m})^m$ .

(4) This inequality is elementary calculus: the minimum of the LHS over all  $\Delta_1$  is achieved for  $1 - \Delta_1 = (\frac{m}{m+1})^m$  and, after substituting this value, it reduces to  $1 - (\frac{m}{m+1})^{m+1} \geq 1 - \frac{1}{e}$ .

□

**Acknowledgments.** We are very grateful to professor Jiří Sgall from the Mathematical Institute of the Academy of Sciences of the Czech Republic, for allowing us to include his proof of Lemma 5.

## References

1. Ahuja, R.K., T.L. Magnanti, and J.B. Orlin [1993], *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall.
2. Gaur, D., T. Ibaraki, and R. Krishnamurti [2002], Constant ratio approximation algorithms for the rectangle stabbing problem and the rectilinear partitioning problem, *Journal of Algorithms* **43**, 138–152.
3. Hassin, R., and N. Megiddo [1991], Approximation algorithm for hitting objects with straight lines, *Discrete Applied Mathematics* **30**, 29–42.
4. Kovaleva, S. [2003], Approximation of Geometric Set Packing and Hitting Set Problems, Ph.D. thesis of Maastricht University.
5. Kovaleva, S., and F.C.R. Spieksma [2002], Primal-dual approximation algorithms for a packing-covering pair of problems, *RAIRO-Operations Research* **36**, 53–72.
6. Vazirani, V.V. [2001], *Approximation Algorithms*, Springer.

# Fast 3-Coloring Triangle-Free Planar Graphs

Lukasz Kowalik\*

Institute of Informatics, Warsaw University  
Banacha 2, 02-097 Warsaw, Poland  
kowalik@mimuw.edu.pl

**Abstract.** We show the first  $o(n^2)$  algorithm for coloring vertices of triangle-free planar graphs using three colors. The time complexity of the algorithm is  $\mathcal{O}(n \log n)$ . Our approach can be also used to design  $\mathcal{O}(n \text{ polylog } n)$ -time algorithms for two other similar coloring problems.

A remarkable ingredient of our algorithm is the data structure processing short path queries introduced recently in [9]. In this paper we show how to adapt it to the fully dynamic environment where edge insertions and deletions are allowed.

## 1 Introduction

The famous Four-Color Theorem says that every planar graph is vertex 4-colorable. The paper of Robertson et al. [10] describes an  $\mathcal{O}(n^2)$  4-coloring algorithm. This seems to be very hard to improve, since it would probably require a new proof of the 4-Color Theorem. On the other hand there are several linear 5-coloring algorithms (see e.g. [4]). Thus efficient coloring planar graphs using only three colors (if possible) seems to be the most interesting area still open for research. Although the general decision problem is NP-hard [6] the renowned Grötzsch's Theorem [8] guarantees that every triangle-free planar graph is 3-colorable. It seems to be widely known that the simplest known proofs by Carsten Thomassen (see [11,12]) can be easily transformed into  $\mathcal{O}(n^2)$  algorithms. In this paper we improve this bound to  $\mathcal{O}(n \log n)$ .

In § 2 we present a new proof of Grötzsch's Theorem, based on a paper of Thomassen [11]. The proof is inductive and it corresponds to a recursive algorithm. The proof is written in a very special way thus it can be immediately transformed into the algorithm. In fact the proof can be treated as a description of the algorithm mixed with a proof of its correctness. In § 3 we discuss how to implement the algorithm efficiently. We describe details of non-trivial operations as well as data structures needed to perform these operations fast. The description and analysis of the most involved one, *Short Path Data Structure* (SPDS), is contained in § 4. This data structure deserves a separate interest. In the paper [9] we presented a data structure built in  $\mathcal{O}(n)$  time and enabling finding shortest paths between given pairs of vertices in constant time, provided that the distance between the vertices is bounded. In our coloring algorithm we need to find paths only of length 1 and 2. Hence here we show a simplified version of the previous, general structure. The SPDS is described from the scratch in order to make this paper

---

\* The research has been supported by KBN grant 4T11C04425. A part of the research was done during author's stay at BRICS, Aarhus University, Denmark.

self-contained but also because we need to introduce some modifications and extensions, like insert and identify operations, not described in our previous paper [9]. We claim that techniques from the present paper can be extended to the general case making possible to use the general data structure in the fully dynamic environment. The description of this extension is beyond the scope of this paper and is intended to appear in the journal version of the paper [9].

Very recently, two new Grötzsch-like theorems appeared. The first one says that any planar graph without triangles at distance less than 4 and without 5-cycles is 3-colorable (see [2]). The other theorem states that planar graphs without cycles of length from 4 to 7 are 3-colorable (see [1]). We claim that our techniques can be adapted to transform these proofs to  $\mathcal{O}(n \log^5 n)$  and  $\mathcal{O}(n \log^7 n)$  algorithms respectively (a careful analysis would probably help to lower these bounds). However, we do not show it in the present paper, since it involves the general short path structure.

**Terminology.** We assume the reader is familiar with standard terminology and notation concerning graph theory and planar graphs in particular (see e.g. [13]). Let us recall here some notions that are not so widely used. Let  $f$  be a face of a connected plane graph. A *facial walk*  $w$  corresponding to  $f$  is the shortest closed walk induced by all edges incident with  $f$ . If the boundary of  $f$  is a cycle the walk is called a *facial cycle*. The length of walk  $w$  is denoted by  $|w|$ . The length of face  $f$  is denoted by  $|f|$  and equal  $|w|$ . Let  $C$  be a simple cycle in a plane graph  $G$ . The length of  $C$  will be denoted by  $|C|$ . The cycle  $C$  divides the plane into two disjoint open domains,  $D$  and  $E$ , such that  $D$  is homeomorphic to an open disc. The set consisting of all vertices of  $G$  belonging to  $D$  and of all edges crossing this domain is denoted by  $\text{int } C$ . Observe that  $\text{int } C$  is not necessarily a graph, while  $C \cup \text{int } C$  is a subgraph of  $G$ . A  $k$ -path ( $k$ -cycle,  $k$ -face) refers to a path (cycle, face) of length  $k$ .

## 2 A Proof of Grötzsch's Theorem

In this section we give a new proof of Grötzsch's Theorem. The proof is based on ideas of C. Thomassen [11]. The reason for writing the new proof is that the original one corresponds to an  $\mathcal{O}(n \log^3 n)$  algorithm when we employ our algorithmic techniques presented in the following sections. In the algorithm corresponding to the proof presented below we don't need to search for paths of length 3 and 4, which reduces the time complexity to  $\mathcal{O}(n \log n)$ . We could also use the recent proof of Thomassen [12] but we suspect that the resulting algorithm would be more complicated and harder to describe. We will need a following lemma that can be easily proved using discharging technique. Due to space limitations we omit the proof.

**Lemma 1.** *Let  $G$  be a biconnected plane graph with every inner face of length at least 5, and the outer face  $C$  of length  $4 \leq |C| \leq 6$ . Furthermore assume that every vertex not in  $V(C)$  has degree at least 3 and that there is no pair of adjacent vertices of degree 2. Then  $G$  has a facial cycle  $C'$  such that  $V(C') \cap V(C) = \emptyset$  and all the vertices of  $C'$  are of degree 3 except, possibly one of degree at most 5.*

Instead of proving Grötzsch's Theorem it will be easier for us to show the following more general result. (Let us note that it follows also from Theorem 5.3 in [7]). A 3-co-

loring of a cycle  $C$  is called *safe* if  $|C| < 6$  or the sequence of successive colors on the cycle is neither  $(1, 2, 3, 1, 2, 3)$  nor  $(3, 2, 1, 3, 2, 1)$ .

**Theorem 1.** *Any connected triangle-free plane graph  $G$  is 3-colorable. Moreover, if the boundary of the outer face of  $G$  is a cycle  $C$  of length at most 6 then any safe 3-coloring of  $G[V(C)]$  can be extended to a 3-coloring of  $G$ .*

*Proof.* The proof is by the induction on  $|V(G)|$ . We assume that  $G$  has at least one uncolored vertex, for otherwise there is nothing left to do. We are going to consider several cases. After each case we assume that none of the previously considered cases applies to  $G$ .

*Case 1.*  $G$  has an uncolored vertex  $x$  of degree at most 2. Then we can remove  $x$  and easily complete the proof by induction. If  $x$  is a cutvertex the induction is applied to each of the connected components of the resulting graph.

*Case 2.*  $G$  has a vertex  $x$  joined to two or three colored vertices. Note that if  $x$  has 3 colored neighbors then  $|C| = 6$  and the neighbors cannot have 3 different colors, as the coloring of  $C$  is safe. Thus we extend the 3-coloring of  $C$  to a 3-coloring of  $G[V(C) \cup \{x\}]$ . Note that for every facial cycle of  $G[V(C) \cup \{x\}]$  the resulting 3-coloring is safe. Then we can apply induction to each face of  $G[V(C) \cup \{x\}]$ .

*Case 3.*  $C$  is colored and has a chord. We proceed similarly as in Case 2.

*Case 4.*  $G$  has a facial walk  $C' = x_1x_2 \cdots x_kx_1$  such that  $k \geq 6$  and at least 1 vertex of  $C'$  is uncolored. As Case 2 is excluded we can assume that  $x_1, x_2$  are uncolored.

*Case 4a.* Assume that  $x_3$  is colored and  $x_1$  has a colored neighbor  $z$ . As Case 2 is excluded,  $x_2$  and  $x_1$  have no colored neighbors, except for  $x_3$  and  $z$  respectively. Then  $G[V(C) \cup \{z, x_1, x_2\}]$  has precisely two inner faces, each of length at least 4. Let  $C_1$  and  $C_2$  denote the facial cycles corresponding to these faces and let  $|C_1| \leq |C_2|$ . We see that  $|C_1| \leq 6$ , because otherwise  $|C| \geq 7$  and  $C$  is not colored. We can assume that  $C_1$  is separating for otherwise  $C_1 = C'$ ,  $|C_1| = 6$ ,  $|C_2| = 6$  and  $C_2$  is separating. Let  $G' = G - \text{int}(C_1)$ . Observe that if cycle  $C_1$  is of length 6 we can add an edge to  $G'$  joining  $x_1$  and the vertex  $w$  at distance 3 from  $x_1$  in  $C_1$  without creating a triangle. Then we can apply the induction hypothesis to  $G'$ . Note that the resulting 3-coloring of  $C_1$  is safe because when  $|C_1| = 6$  vertices  $x_1$  and  $w$  are adjacent in  $G'$ . Moreover, as  $C_1$  is chordless in  $G$ , it is also a 3-coloring of  $G[V(C_1)]$  so we can use induction and extend the coloring to  $C_1 \cup \text{int}(C_1)$ .

*Case 4b.* There is a path  $x_1y_1y_2x_3$  or  $x_1y_1x_3$  distinct from  $x_1x_2x_3$ . Since  $G$  does not contain a triangle,  $x_2 \notin \{y_1, y_2\}$ . Let  $C''$  be the cycle  $x_3x_2x_1y_1y_2x_3$  or  $x_3x_2x_1y_1x_3$  respectively. Let  $G_1 = G - \text{int}(C'')$ . Then  $|V(G_1)| < |V(G)|$  because  $\deg_G(x_2) \geq 3$ . By the induction hypothesis, the 3-coloring of  $C$  can be extended to a 3-coloring of  $G_1$ . The resulting 3-coloring of  $C''$  is also a safe 3-coloring of  $G[V(C'')]$ , since  $|C''| \leq 5$  and  $C''$  is chordless. Thus we can use induction to find a 3-coloring of  $C'' \cup \text{int}(C'')$ .

*Case 4c.* Since Cases 4a and 4b are excluded, we can identify  $x_1$  and  $x_3$  without creating a chord in  $C$  or a triangle in the resulting graph  $G'$ . Hence we can apply the induction hypothesis to  $G'$ .

*Case 5.*  $G$  has a facial cycle  $C'$  of length 4. Furthermore if  $C$  is colored assume that  $C' \neq C$ . Observe that  $C'$  has two opposite vertices  $u, v$  such that one of them is not colored and identifying  $u$  with  $v$  does not create an edge joining two colored vertices. For otherwise, there is a triangle in  $G$  or either of cases 2, 3 occurs.

*Case 5a.* There is a path  $uy_1v$  or  $uy_1y_2v$ ,  $y_1 \notin V(C')$ . Then  $y_2 \notin V(C')$ , for otherwise there is a triangle in  $G$ . Then the path together with one of the two  $u, v$ -paths contained in  $C'$  creates a separating cycle  $C''$  of length 4 or 5 respectively. Then we can apply induction as in Case 4b.

*Case 5b.* Since case 5a is excluded, we can identify  $u$  and  $v$  without creating a triangle or a multiple edge. Thus it suffices to apply induction to the resulting graph. Observe that when  $C$  was colored and  $|C| = 6$  then the coloring of the outer cycle remains safe.

*Case 6 (main reduction).* Observe that since inner faces of  $G$  have length 5 and  $G$  is triangle-free,  $G$  is biconnected. Moreover there is no pair of adjacent vertices of degree 2, for otherwise the 5-face containing this pair contains either a vertex joined to two colored vertices or a chord of  $C$  (Case 2 or 3 respectively). Hence by Lemma 1 there is a face  $C' = x_1x_2x_3x_4x_5x_1$  in  $G$  such that  $\deg(x_1) = \deg(x_2) = \deg(x_3) = \deg(x_4) = 3$ ,  $\deg(x_5) \leq 5$  and  $V(C) \cap V(C') = \emptyset$ . Then vertices  $x_i$  are uncolored. Let  $y_i$  be the neighbor of  $x_i$  in  $G - C'$ , for  $i = 1, 2, 3, 4$ . Moreover, let  $y_5, \dots, y_m$  be the neighbors of  $x_5$  in  $G - C'$ .

*Case 6a.*  $y_i = y_j$  for  $i \neq j$ . Since  $G$  is triangle-free  $x_i$  and  $x_j$  are at distance 2 in  $C'$ . Then there is a 4-cycle  $x_iy_ix_jzx_i$  in  $G$ . As Case 5 is excluded the cycle is separating and we proceed as in Case 4a.

*Case 6b.*  $y_iy_j \in E(G)$  for  $i \neq j$ . Then there is a separating cycle of length 4 or 5 in  $G$  and we proceed as in Case 4a again.

*Case 6c.* There are three distinct vertices  $x_i, x_j, x_k \subset \{x_1, \dots, x_5\}$  such that each has a colored neighbor. Then at least one of cycles in  $G[V(C) \cup \{x_i, x_j, x_k\}]$  is a separating cycle in  $G$  of length from 4 to 6. Then we can proceed exactly as in Case 4a. By symmetry we can assume that  $y_1$  or  $y_2$  is not colored (i.e. we change denotations of vertices  $x_1, \dots, x_4$  and  $y_1, \dots, y_4$ , if needed).

*Case 6d.*  $y_i, y_j$  and  $z$  are colored and  $z$  is a neighbor of  $y_k$  for distinct  $i, j, k$ . Moreover,  $y_i$  and  $z$  have the same color and  $x_i$  is adjacent with  $x_k$ . Again one can see that at least one of cycles in  $G[V(C) \cup V(C') \cup \{y_k\}]$  is a separating cycle in  $G$  of length from 4 to 6 so we can proceed as in Case 4a.

*Case 6e.*  $y_2$  and a neighbor of  $y_1$  have the same color (or  $y_1$  and a neighbor of  $y_2$  have the same color). As Case 6d is excluded  $y_3$  and  $y_4$  are uncolored. By symmetry we can assume that identifying  $y_1$  and  $y_2$  does not introduce an edge with both ends of the same color (i.e. we change denotations of vertices  $x_1, \dots, x_4$  and  $y_1, \dots, y_4$ , if needed).

*Case 6f.*  $y_3$  is colored and  $x_5$  has a colored neighbor. As Cases 6c and 6d are excluded  $y_2$  is uncolored and  $y_4$  has no neighbor with the same color as  $y_3$ . By symmetry we can assume that identifying  $y_1$  with  $y_2$  and  $y_3$  with  $x_5$  does not introduce an edge with both ends of the same color.

*Case 6g.* There is a path  $y_1w_1w_2y_2$  distinct from  $y_1x_1x_2y_2$ . Then we consider a cycle  $C' = y_1w_1w_2y_2x_2x_1y_1$ . As  $G$  is triangle-free,  $\deg x_1 = \deg x_2 = 3$  and  $y_1y_2 \notin E(G)$  cycle  $C'$  has no chord. Case 4 is excluded so  $C'$  is a separating cycle and we can proceed similarly as in Case 4a. The only difference is that this time if  $|C'| = 6$  we try to join  $x_2$  and  $w_1$  to assure that a safe coloring of  $C'$  is obtained. We cannot do it when there is a 2-path in  $G - \text{int}(C')$  between the vertices we want to join because it would create a triangle. Since  $\deg x_2 = 3$  this 2-path is  $x_2x_3w_1$ . But then 5-cycle  $x_2x_3w_1y_1x_1x_2$  is separating and we proceed as in Case 4a.



*Case 6h.* There is a path  $x_5y_kwy_3$  for some  $k \in \{5, \dots, m\}$ . Then we consider a cycle  $C' = y_kx_5x_4x_3y_3wy_k$  and proceed similarly as in Case 6g.

*Case 6i.* Let  $G'$  be the graph obtained from  $G$  by deleting  $x_1, x_2, x_3, x_4$  and identifying  $x_5$  with  $y_3$  and  $y_1$  with  $y_2$ . Since we excluded cases 6c–6h,  $G'$  is triangle-free and the graph induced by colored vertices of  $G'$  is properly 3-colored. Thus we can use induction to get a 3-coloring  $c$  of  $G'$ . We then extend  $c$  to a 3-coloring of  $G$ . As Case 6b is excluded, the (partial) coloring inherited from  $G'$  is proper. If  $c(y_1) = c(x_5)$  then we color  $x_4, x_3, x_2, x_1$ , in that order, always using a free color. If  $c(y_1) \neq c(x_5)$  we put  $c(x_2) = c(x_5)$  and color  $x_4, x_3, x_1$  in that order. This completes the proof.  $\square$

### 3 An Algorithm

The proof of Grötzsch's theorem presented in § 2 can be treated as a scheme of an algorithm. As the proof is inductive, the most natural approach suggests that the algorithm should be recursive. In this section we describe how to implement efficiently the recursive algorithm arising from the proof. In particular we need to explain how to recognize successive cases and how to perform relevant reductions efficiently. We start from describing data structures used by our algorithm. Then we discuss how to use recursion efficiently and how to implement some non-trivial operations of the algorithm. Throughout the paper  $G$  refers to the graph given in the input of our recursive algorithm and  $n$  denotes the number of its vertices.

#### 3.1 Data Structures

**Input Graph, Adjacency Lists.** W. l. o. g. we can assume that the input graph is connected, for otherwise the algorithm is executed separately in each connected component. Moreover, the input graph is given in the form of adjacency lists. We also assume that there is given a planar embedding of the graph, i. e. neighbors of each vertex appear in the relevant adjacency list in the clockwise order given by the embedding.

**Faces and Face Queues.** Observe that using a planar embedding stored in adjacency lists we can easily compute the faces of the input graph. As the graph is connected each face corresponds to a certain facial walk. For every edge  $uv$  there are at most two faces adjacent to  $uv$ . A face is called *right face adjacent to  $(u, v)$*  when  $v$  succeeds  $u$  in the sequence of successive vertices of the facial walk corresponding to the face given in the clockwise order. Otherwise the face is called *left face adjacent to  $(u, v)$* . Each face  $f$  is stored as the corresponding facial walk, i. e. a list of pointers to successive adjacency lists elements corresponding to the edges of the walk. For each such element  $e$  corresponding to neighbor  $v$  of vertex  $u$ , face  $f$  is the right face adjacent to  $(u, v)$ . Additionally,  $e$  stores a pointer to  $f$ . Each face stores also its length, i.e. the length of the corresponding facial walk.

We will also use three queues  $Q_4, Q_5, Q_{\geq 6}$  storing faces of length 4, 5, and  $\geq 6$  respectively, satisfying conditions described in cases 5, 6, 4 of the proof of Theorem 1, respectively.

**Low Degree Vertices Queue.** In order to recognize Case 1 fast we maintain a queue storing the vertices of degree at most 2.

**Short Path Data Structure (SPDS).** In order to search efficiently for 2-paths joining a given pair of vertices we maintain the Short Path Data Structure described in § 4.

### 3.2 Recursion

Note that in the recursive algorithm induced by the proof given in § 2 we need to split  $G$  into two parts. Then each of the parts is processed separately by successive recursive calls. By splitting the graph we mean splitting the adjacency lists and all the other data structures described in the previous section. As the worst-case depth of the recursion is  $\Theta(n)$  the naïve approach would involve  $\Theta(n^2)$  total time spent on splitting the graph. Instead, before splitting the information on  $G$  our algorithm finds the smaller of the two parts. It can be easily done using two DFS calls run in parallel in each of the parts, i.e. each time we find a new vertex in one part, we suspend the search in this part and continue searching in the another. Such an approach finds the smaller part  $A$  in linear time with respect to the size of  $A$ . The other part will be denoted by  $B$ . Then we can easily split adjacency lists, face queues and low degree vertices query. The vertices of the separating cycle (or path) are copied and the copies are added to  $A$ . Note that there are at most 6 such vertices. Next, we delete all the vertices of  $V(A)$  from the SPDS. We will refer to this operation as Cut. As a result of Cut we obtain an SPDS for  $B$ . A Short Path Data Structure for  $A$  is computed from the scratch. In § 4 we show that deletion of an edge from the SPDS takes  $\mathcal{O}(1)$  time and the new SPDS can be built in  $\mathcal{O}(|A|)$  time. Thus the splitting is performed in  $\mathcal{O}(|V(A)|)$  worst-case time.

**Proposition 1.** *The total time spent by the algorithm on splitting data structures before recursive calls is  $\mathcal{O}(n \log n)$*

*Proof.* We can assume that each time we split the graph into two parts – the possible split into three ones described in Case 2 is treated as two successive splits. Let us call the vertices of the separating cycle (or path) as *outer vertices* and the remaining ones from the smaller part  $A$  as *inner vertices*. The total time spent on splitting data structures is linear with the total number of inner and outer vertices. As there are  $\mathcal{O}(n)$  splits, and each split involves at most 6 outer vertices the total number of outer vertices to be considered is  $\mathcal{O}(n)$ . Moreover, as during each split of a  $k$ -vertex graph there are at most  $\lfloor \frac{k}{2} \rfloor$  inner vertices each vertex of the input graph becomes an inner vertex at most  $\log n$  times. Hence the total number of inner vertices is  $\mathcal{O}(n \log n)$ .  $\square$

### 3.3 Non-trivial Operations

**Identifying Vertices.** In this section we describe how our algorithm updates the data structures described in § 3.1 during the operation of identifying a pair of vertices  $u, v$ . Identifying two vertices can be performed using deletions and insertions. More precisely, the operation  $\text{Identify}(u, v)$  is executed using the following algorithm. First it compares degrees of  $u$  and  $v$  in graph  $G$ . Assume that  $\deg_G(u) \leq \deg_G(v)$ . Then for each neighbor  $x$  of  $u$  we delete edge  $ux$  from  $G$  and add a new edge  $vx$ , unless it is already present in the graph.

**Lemma 2.** *The total number of pairs of delete/insert operations performed by Identify algorithm is bounded by  $\mathcal{O}(n \log n)$ .*

*Proof.* For now, assume that there are no other edge deletions performed by our algorithm, except for those involved with identifying vertices. The operation of deleting edge  $ux$  and adding  $vx$  during  $\text{Identify}(u, v)$  will be called *moving edge  $ux$* . We see that each edge of the input graph can be moved at most  $\lceil \log n \rceil$  times, for otherwise there would appear a vertex of degree  $> n$ . Subsequently, there are  $\mathcal{O}(n \log n)$  pairs of delete/insert operations performed during Identify operation. It is clear that this number does not increase when we consider additional deletions.  $\square$

As we always identify a pair of vertices in the same facial walk it is straightforward to update adjacency lists. Lemma 2 shows that we need  $\mathcal{O}(n \log n)$  time in total for these updates including updating the information about the faces incident to  $x$  and  $y$ . Each of affected faces is then placed in appropriate face queue (if one has to be changed). In § 4 we show how to update the Short Path Data Structure efficiently after Identify. To sum up, identifying vertices takes  $\mathcal{O}(n \log n)$  time including updating data structures.

**Finding Short Paths.** In our algorithm we need to find paths of length 1, 2 or 3 between given pairs of vertices. Observe that there are  $\mathcal{O}(n)$  such queries during an execution of the whole algorithm. As we show in § 4 paths of length 1 or 2 can be found in  $\mathcal{O}(1)$  time using the Short Path Data Structure. It remains to focus on paths of length 3.

Let us describe an algorithm Path3 that will be used to find a 3-path between a pair of vertices  $u, v$ , if there is any. We can assume that we are given a path  $p$  of length 2 (cases 4b and 5a) or of length 3 (Case 6g) joining  $u$  and  $v$ . W.l.o.g. we assume that  $\deg(u) \leq \deg(v)$ . Let  $A(u), A(v)$  denote the adjacency lists of  $u$  and  $v$ , respectively. We start from assigning variables  $x_1$  and  $x_2$  to the element of  $A(u)$  corresponding to the edge of  $p$  incident with  $u$ . Similarly, we assign  $x_3$  and  $x_4$  to the element of  $A(v)$  corresponding to the edge of  $p$  incident with  $v$ . Then we start a loop. We assign  $x_1$  to the succeeding element and  $x_2$  to the preceding element in  $A(u)$ . Similarly, we assign  $x_3$  to the succeeding element and  $x_4$  to the preceding element in  $A(v)$ . Then we use the Short Path Data Structure to search for paths of length 2 between  $x_1$  and  $v$ ,  $x_2$  and  $v$ ,  $x_3$  and  $u$ ,  $x_4$  and  $u$ . If a path is found, the algorithm stops, otherwise we repeat the loop. If no 3-path exists at all, the loop stops when all the neighbors of  $u$  are checked.

**Lemma 3.** *The total time spent on performing Path3 algorithm is  $\mathcal{O}(n \log n)$ .*

*Proof.* We can divide these operations into two groups. The operation is called *successful* if there exists a 3-path joining  $u$  and  $v$ , distinct from  $p$  when  $|p|=3$ , and *failed* in the other case. Recall that when there is no such 3-path, vertices  $u$  and  $v$  are identified. As the time complexity of a single execution of Path3 algorithm is  $\mathcal{O}(\deg u)$  and all the edges incident with  $u$  are deleted during  $\text{Identify}(u, v)$  operation, the total time spent on performing failed Path3 operations is linear in the number of edge deletions caused by identifying vertices. By Lemma 2 there are  $\mathcal{O}(n \log n)$  such edge deletions.

Now it remains to estimate the time used by successful operations. Let us consider one of them. Let  $C'$  be the separating cycle compound of the path  $p$  and the 3-path that was found. Let  $H$  be the graph  $C' \cup \text{int}(C')$ . Recall that one of vertices  $u, v$  is not

colored, say  $v$ . Then the number of queries sent to the SPDS is at most  $4 \cdot \deg_H(v)$ . Note that since  $v$  will be colored in graph  $H$  the total number of queries asked during executions of successful Path3 operations is at most 8 times larger than the total number of edges appearing in  $G$  (an edge  $xv$  added after deleting  $xu$  during  $\text{Identify}(u, v)$  is not counted as a new one here). Thus the time used by successful operations is  $\mathcal{O}(n)$ .  $\square$

**Removing a Vertex of Degree at Most 3.** In cases 1 and 6i we remove vertices of degrees 1, 2 or 3. There are at most  $\mathcal{O}(n)$  such operations in total. As the degrees are bounded the total time spent on updating the Short Path Data Structure and adjacency lists is  $\mathcal{O}(n)$ . We also need to update information about incident faces. We may need to join two or three of them. It is easy to update the facial walk of the resulting face in  $\mathcal{O}(1)$  time by joining the walks of the faces incident to the deleted vertex. The problem is that edges contained in the walk corresponding to the new face store pointers to two different faces. Thus we use the well-known Find and Union algorithm (see e.g. [5]) for finding a right face adjacent to a given edge. The amortized time of the search is  $\mathcal{O}(\log^* n)$ . As the total number of all these searches is  $\mathcal{O}(n)$  it does not increase the overall time complexity of our coloring algorithm.

Before deleting an edge we additionally need to check whether it is a bridge. The check can be easily done by verifying whether the edge has the right face equal to its left face. If so, after deleting the edge the face is split into two faces in  $\mathcal{O}(1)$  time and we process each of the connected components recursively.

**Searching for Faces.** To search for the faces described in the cases 5, 6, 4 of the proof from § 2 we use queues  $Q_4, Q_5, Q_{\geq 6}$ , respectively. The queues are initialized in  $\mathcal{O}(n)$  time and the searches are performed in  $\mathcal{O}(1)$  time.

**Additional Remarks.** To recognize cases 2, 3, 4a, 6c–6f efficiently it suffices to pass down the outer cycle in the recursion, when the cycle is colored. Then it takes only  $\mathcal{O}(1)$  time to recognize each case (in some cases we use Short Path Data Structure) since there are at most 6 colored vertices.

## 4 Short Path Data Structure

In this section we describe the *Short Path Data Structure* (SPDS) which can be built in linear time and enables finding shortest paths of length at most 2 in planar graphs in  $\mathcal{O}(1)$  time. Moreover, we show here how to update the structure after deleting an edge, adding an edge and after identifying a pair of vertices. Then we analyze the total time needed for updates of the SPDS during the particular sequence of operations appearing in our 3-coloring algorithm. The time turns out to be bounded by  $\mathcal{O}(n \log n)$ .

### 4.1 The Structure and Processing the Queries

The Short Path Data Structure consists of two elements, denoted as  $\overrightarrow{G_1}$  and  $\overrightarrow{G_2}$ . We will describe them after introducing some basic notions.

A directed graph is said to be  $k$ -oriented if its every vertex has the out-degree at most  $k$ . If one can orient edges of an undirected graph  $H$  obtaining  $k$ -oriented graph

$H'$  we say that  $H$  can be  $k$ -oriented. In particular, when  $k = \mathcal{O}(1)$  we will say that  $H'$  is  $\mathcal{O}(1)$ -oriented and  $H$  can be  $\mathcal{O}(1)$ -oriented.  $\vec{H}$  will denote certain orientation of a graph  $H$ . The *arboricity* of a graph  $H$  is the minimal number of forests needed to cover all the edges of  $H$ . Observe that a graph with arboricity  $a$  can be  $a$ -oriented.

**Graph  $\vec{G}_1$  and Adjacency Queries.** In this section  $G_1$  denotes a planar graph for which we build a SPDS (recall from § 3.2 that it is not only the input graph). It is widely known that planar graphs have arboricity at most 3. Thus  $G_1$  can be  $\mathcal{O}(1)$ -oriented. Let  $\vec{G}_1$  denote such an orientation of  $G_1$ . Then  $xy \in E(G_1)$  iff  $(x, y) \in E(\vec{G}_1)$  or  $(y, x) \in E(\vec{G}_1)$ . Thus, providing that we can maintain bounded out-degrees in  $\vec{G}_1$  during our coloring algorithm, we can process in  $\mathcal{O}(1)$  time the queries of the form: “Are vertices  $x$  and  $y$  adjacent?”.

**Graph  $\vec{G}_2$ .** Let  $G_2$  be a graph with the same vertex set as  $G_1$ . Moreover, edge  $vw$  is in  $G_2$  iff there exists vertex  $x \in V(\vec{G}_1)$  such that  $(x, v) \in E(\vec{G}_1)$  and  $(x, w) \in E(\vec{G}_1)$ . Vertex  $x$  is said to *support* edge  $vw$ . Since  $\vec{G}_1$  has bounded out-degree every vertex supports  $\mathcal{O}(1)$  edges in  $G_2$ . Hence  $G_2$  is of linear size. The following lemma states even more (proof is omitted due to space limitations):

**Lemma 4.** *Let  $\vec{G}_1$  be a directed planar graph with out-degree bounded by  $d$ . Let  $G_2$  be an undirected graph with  $V(G_2) = V(\vec{G}_1)$  and  $E(G_2) = \{vw : (x, v) \in E(\vec{G}_1) \text{ and } (x, w) \in E(\vec{G}_1)\}$ . Then  $G_2$  is a union of at most  $4 \cdot \binom{d}{2}$  planar graphs.*

**Corollary 1.** *If the out-degree in graph  $\vec{G}_1$  is bounded by  $d$  then graph  $G_2$  has arboricity bounded by  $12 \cdot \binom{d}{2}$ .*

**Corollary 2.** *Graph  $G_2$  can be  $\mathcal{O}(1)$ -oriented.*

Corollary 1 follows immediately since the arboricity of a planar graph is at most 3. By  $\vec{G}_2$  we will denote an  $\mathcal{O}(1)$ -orientation of  $G_2$ . Let  $e$  be an edge in  $\vec{G}_2$  equal  $(v, w)$  or  $(w, v)$ . Let  $x$  be a vertex that supports  $e$  and let  $e_1 = (x, v)$  and  $e_2 = (x, w)$ ,  $e_1, e_2 \in E(\vec{G}_1)$ . We say that edges  $e_1$  and  $e_2$  are *parents* of  $e$  and  $e$  is a *child* of  $e_1$  and  $e_2$ . We say that a pair  $\{e_1, e_2\}$  is a *couple of parents* of  $e$ . Notice that each edge can have more than one couple of parents. We additionally store the following information:

- for each  $e \in E(\vec{G}_2)$  a list  $P(e)$  of all pairs  $\{e_1, e_2\}$  such that  $e$  is a common child of  $e_1$  and  $e_2$ ,
- for each  $e \in E(\vec{G}_1)$  a list  $C(e)$  of pairs  $(c, p)$  where  $c \in E(\vec{G}_2)$  is a common child of  $e$  and certain edge  $f$  and  $p$  is a pointer to  $\{e, f\}$  in the list  $P(c)$ .

**Queries About 2-Paths.** It is easy to see that when  $\vec{G}_1$  and  $\vec{G}_2$  are  $\mathcal{O}(1)$ -oriented we can find a path  $uxv$  of length 2 joining a pair of given distinct vertices  $u, v$  in  $\mathcal{O}(1)$  time as follows:

- (i) check whether there is an oriented path  $uxv$  or  $vxu$  in  $\vec{G}_1$ ,
- (ii) check whether there is a vertex  $x$  such that  $(u, x), (v, x) \in E(\vec{G}_1)$ ,

- (iii) check whether there is an edge  $e = (u, v)$  or  $e = (v, u)$  in  $\vec{G}_2$ . If so, pick any of its couples of parents  $\{(x, u), (x, v)\}$  stored in  $P(e)$ .

To build the shortcut graph data structure for a given graph  $G_1$ , namely graphs  $\vec{G}_1$  and  $\vec{G}_2$ , we start from creating two graphs containing the same vertices as  $G_1$  but no edges. Then for every edge  $uv$  from  $G_1$  we add  $uv$  to SPDS using insertion algorithm described in the following section. In § 4.3 we show that it takes only  $\mathcal{O}(|V(G_1)|)$  time.

## 4.2 Inserting and Deleting Edges

**Maintaining Bounded Out-degrees in  $\vec{G}_1$  and  $\vec{G}_2$ .** As graph  $G_1$  is dynamically changing we will need to add and remove edges from  $\vec{G}_1$  and  $\vec{G}_2$ . While removing is easy, after adding an edge we may need to reorient some edges to leave the graph  $\mathcal{O}(1)$ -oriented. We use the approach of G. Brodal and R. Fagerberg [3]. Assume that  $H$  is an arbitrary graph of arboricity  $a$ . Let  $\Delta = 6a - 1$  and assume that we want to maintain a  $\Delta$ -orientation of  $G$ , denoted by  $\vec{H}$ . They consider the following routine for inserting an edge  $uv$ . First add  $(u, v)$  to  $\vec{H}$ . If  $\text{outdeg } u = \Delta + 1$ , repeatedly a node  $w$  with out-degree larger than  $\Delta$  is picked, and the orientation of all the edges outgoing from  $w$  is changed. Deleting an edge does not need any reorientations. We will refer to these routines as *Insert*  $(u, v)$  and *Delete*  $(u, v)$  respectively and we will use them for inserting and deleting edges in  $\vec{G}_1$  and  $\vec{G}_2$ . Observe that the out-degrees in  $\vec{G}_1$  will be bounded by 17 since it has arboricity 3 as a planar graph. Subsequently Corollary 1 guarantees that  $\vec{G}_2$  has bounded arboricity and hence it will be also  $\mathcal{O}(1)$ -oriented.

**Updating the SPDS After a Deletion.** Note that after deleting an edge from  $\vec{G}_1$  we need to find out which edges in  $\vec{G}_2$  should be deleted, if any. Assume that  $e$  is an edge of  $\vec{G}_1$  and it is going to be deleted. For each pair  $(c, p) \in C(e)$  we have to perform the following operations: remove the pair  $\{e, f\}$  referenced by pointer  $p$  from list  $P(c)$ , remove the pair  $(c, p)$  from the list  $C(f)$ . If list  $P(c)$  becomes empty we delete edge  $c$  from  $G_2$ . Since  $e$  has at most  $d = \mathcal{O}(1)$  children, the following proposition holds:

**Proposition 2.** *After deletion of an edge the SPDS can be updated in  $\mathcal{O}(1)$  time.*

**Updating the SPDS After an Insertion.** To perform insertion we first call *Insert* in graph  $\vec{G}_1$ . Whenever any edge in  $\vec{G}_1$  changes its orientation we act as if it was deleted and update  $\vec{G}_2$  as described above. Moreover, when any edge  $(u, v)$  appears in  $\vec{G}_1$ , both after *Insert*  $(u, v)$  and after reorienting  $(v, u)$ , we add an edge  $vw$  to  $G_2$  for each edge  $(u, w)$  present in  $\vec{G}_1$ .

## 4.3 Time Complexity of Updating the SPDS

There are four operations performed by our coloring algorithm on an input graph. First two are insertions and *Identify* operations. The third one is deleting a vertex of degree at most 3 and will be denoted as *DeleteVertex*. The last operation is *Cut* described in § 3.2. These four operations will be called *meta-operations*. Each of them will be performed using a sequence of deletions and insertions. In this section we show that for the particular sequence of meta-operations appearing in our coloring algorithm the

total time needed for updating the SPDS is  $\mathcal{O}(n \log n)$ . The following lemma is a slight generalization of Lemma 1 from the paper [3]. Their proof remains valid even for the modified formulation presented below.

**Lemma 5.** *Given an arboricity preserving sequence  $\sigma$  of edge insertions and deletions on an initially empty graph, let  $H_i$  be the graph after the  $i$ -th operation.*

*Let  $\vec{H}_1, \vec{H}_2, \dots, \vec{H}_{|\sigma|}$  be any sequence of (2a)-orientations with at most  $r$  edge reorientations in total and let  $\vec{F}_1, \vec{F}_2, \dots, \vec{F}_{|\sigma|}$  be the  $\Delta$ -orientations of  $H_1, H_2, \dots, H_{|\sigma|}$  appearing when Insert and Delete algorithms are used to perform  $\sigma$ . Let  $k$  be the number of edges  $uv$  such that  $(u, v) \in \vec{F}_i$ ,  $(v, u) \in \vec{H}_i$  and the  $i$ -th operation in  $\sigma$  is insertion of  $uv$ .*

*Then the Insert and Delete algorithms perform at most  $3(k + r)$  edge reorientations in total on the sequence  $\sigma$ .*

**Lemma 6.** *The total number of reorientations in  $\vec{G}_1$  used by Insert and Delete algorithms to perform a sequence of  $k$  meta-operations leading to the empty graph is  $\mathcal{O}(k)$ .*

*Proof.* Let us denote the sequence of meta-operations by  $\sigma$ . Let  $G^i$  be the graph  $G_1$  after the  $i$ -th meta-operation. We will construct a sequence of 6-orientations  $\mathcal{G} = \vec{G}^1, \vec{G}^2, \dots, \vec{G}^k$ . Observe that  $\vec{G}^k$  has no edges thus it is 6-oriented. We will describe  $\vec{G}^i$  using  $\vec{G}^{i+1}$ . If  $\sigma_i$  is an insertion of edge  $uv$ ,  $\vec{G}^i$  is obtained from  $\vec{G}^{i+1}$  by deleting  $uv$ . If  $\sigma_i = \text{Identify}(u, v)$  the edges incident with the identified vertex  $x$  in  $\vec{G}^{i+1}$  are partitioned into two groups in  $\vec{G}^i$  without changing their orientation. Recall that  $u$  and  $v$  are not adjacent in  $\vec{G}^i$ . Thus  $\text{outdeg}_{\vec{G}^i} u \leq \text{outdeg}_{\vec{G}^i} x$  and  $\text{outdeg}_{\vec{G}^i} v \leq \text{outdeg}_{\vec{G}^i} x$ . If  $\sigma_i = \text{DeleteVertex}$  we simply add a vertex of degree at most 3 to  $\vec{G}^i$  in such a way that the added edges leave the new vertex. Its out-degree is 3 and the out-degrees of other vertices do not change. Finally we consider the case when  $\sigma_i$  is Cut operation. Let  $A$  be the graph induced by the vertices removed from  $G^i$ . As  $A$  is planar it can be 3-oriented. The remaining edges, joining a vertex from  $A$ , say  $x$ , with a vertex from  $G^{i+1}$ , say  $y$  are oriented from  $x$  to  $y$ . Observe that a vertex in  $A$  can be adjacent with at most 3 vertices in  $G^{i+1}$ , since there are no triangles. Thus  $\vec{G}^{i+1}$  is 6-oriented. Description of the sequence  $\mathcal{G}$  of orientations is finished now. Observe that there is no single reorientation in this sequence.

Now let us compare the sequence  $\mathcal{G}$  with the sequence  $\mathcal{F}$  of 17-orientations appearing when Insert and Delete algorithms are used to perform  $\sigma$ . Consider insertions involved with a single Identify operation. Then at most  $6 + 17 = \mathcal{O}(1)$  inserted edges obtain different orientations in  $\mathcal{G}$  and  $\mathcal{F}$ . Hence the total number of such insertions involved with identifying vertices is  $\mathcal{O}(k)$ . Operations DeleteVertex and Cut do not use insertions and trivially there are at most  $k$  ordinary insertions in  $\sigma$ . Thus by Lemma 5 the Insert and Delete algorithms perform  $\mathcal{O}(k)$  reorientations in  $\vec{G}_1$  to perform  $\sigma$ .  $\square$

**Lemma 7 (proof omitted).** *Let  $\sigma$  be a sequence of  $k + l$  meta-operations performed on an initially empty planar graph  $G_1 = (V, E)$ . Moreover, let the first  $k$  operations in  $\sigma$  be insertions. Then the total number of reorientations in  $\vec{G}_2$  used by Insert and Delete algorithms to perform  $\sigma$  is  $\mathcal{O}(l \log |V|)$ .*



**Corollary 3.** *For any planar graph  $G_1$  the Short Path Data Structure can be constructed in  $\mathcal{O}(|V(G_1)|)$  time.*

**Corollary 4.** *Let  $n$  be the number of vertices of the graph on the input of the coloring algorithm. The total time needed to perform all meta-operations executed by the coloring algorithm, including updating the Short Path Data Structure, is  $\mathcal{O}(n \log n)$ .*

*Proof.* By Lemma 2 all Identify operations cause  $\mathcal{O}(n \log n)$  pairs of deletions/insertions. By Proposition 1 all Cut operations cause  $\mathcal{O}(n \log n)$  deletions. There is  $\mathcal{O}(n)$  meta-operations performed. Hence, by Lemmas 6 and 7 the total number of reorientations needed by insert operations is  $\mathcal{O}(n \log n)$ . It ends the proof.  $\square$

Let us also note that using techniques from proof of Lemma 7 one can show that update of SPDS after an insertion in *arbitrary* sequence of insert/delete operations takes amortized  $\mathcal{O}(\log^2 n)$  time. The approach presented here can be extended to the general version of SPDS. It needs  $\mathcal{O}(1)$  time to find a shortest path between vertices at distance bounded by a constant  $k$  and updates after an insertion in  $\mathcal{O}(\log^k n)$  amortized time.

**Acknowledgments.** I would like to thank Krzysztof Diks and anonymous referees for reading this paper carefully and helpful comments. Thanks go also to Maciej Kurowski for many interesting discussions in Århus, not only these on Grötzsch's Theorem.

## References

1. O. V. Borodin, A. N. Glebov, A. Raspaud, and M. R. Salavatipour. Planar graphs without cycles of length from 4 to 7 are 3-colorable. 2003. Submitted to J. of Comb. Th. B.
2. O. V. Borodin and A. Raspaud. A sufficient condition for planar graphs to be 3-colorable. *Journal of Combinatorial Theory, Series B*, 88:17–27, 2003.
3. G. S. Brodal and R. Fagerberg. Dynamic representations of sparse graphs. In *Proc. 6th Int. Workshop on Algorithms and Data Structures*, volume 1663 of *LNCS*, pages 342–351. 1999.
4. N. Chiba, T. Nishizeki, and N. Saito. A linear algorithm for five-coloring a planar graph. *J. Algorithms*, 2:317–327, 1981.
5. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT, 2001.
6. M. R. Garey and D. S. Johnson. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, February 1976.
7. J. Gimbel and C. Thomassen. Coloring graphs with fixed genus and girth. *Transactions of the AMS*, 349(11):4555–4564, November 1997.
8. H. Grötzsch. Ein Dreifarbensatz für dreikreisfreie Netze auf der Kugel. Technical report, Wiss. Z. Martin Luther Univ. Halle Wittenberg, Math.-Nat. Reihe 8, pages 109–120, 1959.
9. J. Kowalik and M. Kurowski. Shortest path queries in planar graphs in constant time. In *Proc. 35th Symposium Theory of Computing*, pages 143–148. ACM, June 2003.
10. N. Robertson, D. P. Sanders, P. Seymour, and R. Thomas. Efficiently four-coloring planar graphs. In *Proc. 28th Symposium on Theory of Computing*, pages 571–575. ACM, 1996.
11. C. Thomassen. Grötzsch's 3-color theorem and its counterparts for the torus and the projective plane. *Journal of Combinatorial Theory, Series B*, 62:268–279, 1994.
12. C. Thomassen. A short list color proof of Grötzsch's theorem. *Journal of Combinatorial Theory, Series B*, 88:189–192, 2003.
13. D. West. *Introduction to Graph Theory*. Prentice Hall, 1996.



# Approximate Unions of Lines and Minkowski Sums

Marc van Kreveld and A. Frank van der Stappen

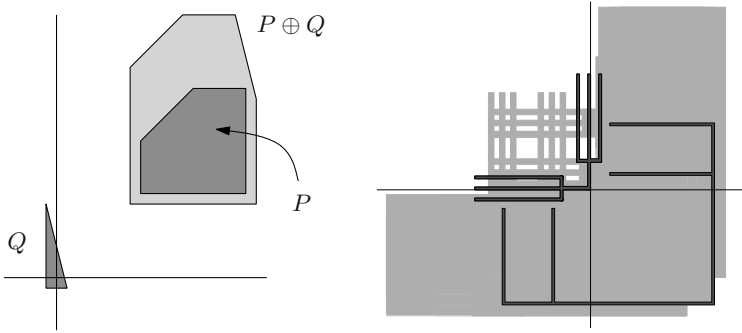
Institute of Information and Computing Sciences, Utrecht University,  
PO Box 80.089, 3508 TB Utrecht, The Netherlands  
{marc, frankst}@cs.uu.nl

**Abstract.** We study the complexity of and algorithms to construct approximations of the union of lines, and of the Minkowski sum of two simple polygons. We also study *thick* unions of lines and Minkowski sums, which are inflated with a small disc. Let  $b = D/\varepsilon$  be the ratio of the diameter of the region of interest and the distance (or error) of the approximation. We present upper and lower bounds on the combinatorial complexity of approximate and thick unions of lines and Minkowski sums, with bounds expressed in  $b$  and the input size  $n$ . We also give efficient algorithms for the computation.

## 1 Introduction

Minkowski sums are a basic concept in motion planning and therefore, they have been studied extensively in computational geometry. For two sets  $P$  and  $Q$  in the plane, it is defined as  $P \oplus Q = \{p + q \mid p \in P, q \in Q\}$ , where  $p$  and  $q$  are points whose coordinates are added, see Figure 1 for an example. Minkowski sums are used to define the free configuration space of a translating robot amidst obstacles [13,19]. For essentially the same reason, Minkowski sums are important in packing problems, where given shapes have to be packed without overlap and without rotation inside a given region. This problem shows up in efficient use of fabric in cloth manufacturing [3,11,15]. Other applications are in cartography [21]. One of the basic questions that arises is: Given two simple polygons with  $n$  vertices in total, compute their Minkowski sum. All algorithms for this problem must take  $\Omega(n^4)$  time, simply because the output can have as large complexity.

In practice, Minkowski sums of high complexity rarely show up, and it is interesting to discover under what conditions the maximum complexity of the Minkowski sum is considerably lower. Similarly, it is interesting to know when we can compute it more efficiently. Questions like these have led to research on *realistic input models* and the development of geometric algorithms for *fat objects*. One of the main results is that the union complexity of  $n$  triangles, all of whose angles are at least some given constant, is  $O(n \log \log n)$  rather than  $\Theta(n^2)$  [14,18]. More recently, Erickson showed that two *local polygons* with  $n$  vertices have a Minkowski sum of complexity  $O(n^3 \log n)$  [9]. Much research has been done on realistic input models, see [4,5,6,7,8,16,20,22] for a sample.



**Fig. 1.** Left: Minkowski sum of  $P$  and  $Q$ . Right: The Minkowski sum of two polygons can have complexity  $\Theta(n^4)$  in the worst case.

This paper also deals with Minkowski sums and ways to overcome the  $\Omega(n^4)$  lower bound for construction. We will not make assumptions on the input polygons, but instead study *approximate* Minkowski sums. We allow the Minkowski sum that we compute to be off by a distance  $\varepsilon$ , and assume that there is a region of interest with diameter  $D$ . We let  $b = D/\varepsilon$  be the relative precision, and we prove complexity and running time bounds for approximate Minkowski sums. We assume throughout the paper that  $b = \Omega(1)$ . If we do not make assumptions on both  $\varepsilon$  and  $D$ , we cannot get any bound better than  $\Theta(n^4)$ .

As a simpler first step we consider approximate unions of lines. We show that a union of  $n$  lines admits an approximation of complexity  $O(\min(b^2, n^2))$ , which is optimal in the worst case, and we can compute it in  $O(n + n^{2/3+\delta}b^{4/3})$  time assuming  $b = O(n)$ , for any  $\delta > 0$ . A special type of approximate union is the union of thick lines, or strips. It is the maximal subset of the plane that is approximate. Being off by a distance at most  $\varepsilon$  implies that the strips have width  $2\varepsilon$ . We show that  $n$  such strips in a region of diameter  $D$  have union complexity  $\Omega(n + b^2)$  and  $O(n + b\sqrt{bn})$  in the worst case, and we can compute it in  $O(n^{4/3+\delta}b^{2/3})$  time assuming  $b = O(n)$ .

Then we proceed with approximate and thick Minkowski sums. For thick Minkowski sums we prove a lower bound of  $\Omega(n^2 + b^2)$  and an upper bound of  $O((n + b)n\sqrt{b\log n} + n^2 \log n)$  on the complexity. The approximation can be computed in  $O(\min\{nb^2, (nb)^{4/3+\delta}\})$  time and the thick Minkowski sum in  $O(n^{8/3+\delta}b^{2/3})$  time.

Interestingly, papers on fatness also imply complexity bounds on thick arrangements and Minkowski sums [7,14,18]. The fatness parameter is related to  $b$  and we would get a bound of  $O(nb \log b)$  for the thick union of lines and  $O(n^2 b^2 \log b)$  for the thick Minkowski sum [18]. The bounds presented in this paper are better for most values of  $b$ : For any  $\gamma > 0$ , we have better bounds if  $b = \Omega(n^\gamma)$  and  $b = O(n^{1-\gamma})$  (thick union of lines) or  $b = O(n^{2-\gamma})$  (thick Minkowski sums).

## 2 Approximate and Thick Sets

We are interested in scenarios in which we do not need the exact shape of the union of lines or Minkowski sum, but instead may settle for a structure that is ‘close’. This freedom allows us to remove holes in the union or Minkowski sum that are smaller than the level of detail that we are interested in, thereby reducing the combinatorial complexity of these structures. The closeness can be expressed by an absolute precision value  $\varepsilon > 0$ . We will study unions of lines and Minkowski sums in this approximate sense.

**Definition 1.** An  $\varepsilon$ -approximation of a set  $C \subseteq \mathbf{R}^2$  is a set  $A \subseteq \mathbf{R}^2$  such that for any point  $p \in A$ , there is a point  $p' \in C$  at a distance at most  $\varepsilon$ , and for any point  $p \in C$ , there is a point  $p' \in A$  at a distance at most  $\varepsilon$ .

We will also study a particular  $\varepsilon$ -approximation. The  $\varepsilon$ -thick set  $C$  is the largest-area  $\varepsilon$ -approximation of  $C$ . Let  $\Delta_\varepsilon$  be the disc of radius  $\varepsilon$  centered at the origin.

**Definition 2.** The  $\varepsilon$ -thick set  $C \subseteq \mathbf{R}^2$  is the Minkowski sum  $C \oplus \Delta_\varepsilon$ .

We are interested in  $\varepsilon$ -approximations of reduced combinatorial complexity. A small absolute precision value  $\varepsilon$ , however, does not lead to a complexity reduction. Let  $I \subseteq \mathbf{R}^2$  be a region of interest, and let  $D$  be its diameter. The interesting cases in terms of complexity reductions occur when the ratio of the diameter  $D$  of  $I$  and the precision value  $\varepsilon$  is bounded. The ratio of  $b = D/\varepsilon$  can be seen as a measure for (the inverse of) the relative precision of the approximation.

## 3 Unions of Lines

Let  $L$  be a set of  $n$  lines. We study the combinatorial complexity of approximate and thick unions of the lines in  $L$ . In addition we present algorithms for computing the approximate and thick unions.

### 3.1 Complexity

The complexity of an arrangement of  $n$  lines is  $O(n^2)$  and this bound is tight. Approximations of lower complexity exist for the union of the  $n$  lines in  $L$  when the relative precision  $b = o(n)$ . We first prove a lower bound.

**Lemma 1.** Any  $\varepsilon$ -approximation of the union of  $n$  lines has complexity  $\Omega(b^2)$  in the worst case, assuming  $b = O(n)$ .

*Proof.* (sketch) Place a set  $L$  of  $O(b)$  horizontal and vertical lines at distance  $6\varepsilon$  to form a regular square grid in  $I$ . This creates  $\Omega(b^2)$  squares of  $4\varepsilon$  by  $4\varepsilon$  in which no point of the approximation may be. We can argue that any (polygonal) approximation must have a vertex close to every one of the  $\Omega(b^2)$  squares.

**Lemma 2.** Any union of  $n$  lines has an  $\varepsilon$ -approximation of size  $O(\min(b^2, n^2))$ .

*Proof.* As the set  $L$  of lines is its own  $\varepsilon$ -approximation, the  $O(n^2)$  bound is trivial. Now consider the  $b^2$  points on a regular  $\varepsilon \times \varepsilon$  grid inside  $I$ . Any point closer than  $\varepsilon$  from some point on a line in  $L$  is chosen to be in the set  $A$ . It is clear that  $A$  is an  $\varepsilon$ -approximation, of complexity  $O(b^2)$ .

The  $\varepsilon$ -thick union of lines equals the union of  $\varepsilon$ -thick lines. Let  $L_\varepsilon = \{l \oplus \Delta_\varepsilon \mid l \in L\}$ . All elements of  $L_\varepsilon$  are strips of width  $2\varepsilon$ .

**Lemma 3.** *The complexity of the  $\varepsilon$ -thick union of  $n$  lines can be  $\Omega(n + b^2)$ .*

*Proof.* Choose  $b/3$  equally-spaced horizontal strips (e.g. a distance  $\varepsilon$  apart) that all intersect  $I$ . Choose  $b/3$  vertical strips in the same way. The complexity of the union of these strips is  $\Omega(b^2)$ . We let  $\Omega(n)$  strips with slightly different positive slopes intersect the uncovered lower right corner of  $I$ , such that each of the  $\Omega(n)$  pairs of strips with successive slopes adds a vertex on an arbitrarily small circular arc centered at the corner.

A hole in the union of strips from  $L_\varepsilon$  originates from a hole with area  $\Omega(\varepsilon^2)$  in the union of the lines from  $L$ . As a result, the number of holes in the union of strips that intersect  $I$  is  $O(b^2)$ . We can now prove:

**Lemma 4.** *The complexity of the  $\varepsilon$ -thick union of  $n$  lines is  $O(n + b\sqrt{bn})$ .*

*Proof.* A vertex  $v$  of the union is the intersection of a line  $\beta_v$  bounding a strip  $s_v \in L_\varepsilon$  and a line  $\beta'_v$  bounding another strip  $s'_v \in L_\varepsilon$ . It is an endpoint of the equally-long segments  $\beta_v \cap s'_v$  and  $\beta'_v \cap s_v$ , whose interiors are in the interior of the union. We separate the vertices  $v$  inside  $I$  for which  $\beta_v \cap s'_v$  intersects the boundary  $\partial I$  of  $I$  from those for which this segment lies entirely inside  $I$ .

If  $\beta_v \cap s'_v$  intersects  $\partial I$ , we charge the vertex  $v$  to the intersection  $\beta_v \cap s'_v \cap \partial I = \beta_v \cap \partial I$ . The intersection will be charged only once, giving  $O(n)$  vertices of this type in total.

We next handle the union vertices  $v$  in  $I$  for which  $\beta_v \cap s'_v$  lies entirely inside  $I$ . Each union vertex  $v$  belongs to a convex hole. We define the (turning) angle  $\alpha_v$  to be  $\pi - \phi$ , where  $\phi$  ( $\phi < \pi$ ) is the angle between the union edges incident to  $v$  (or the equivalent angle between  $\beta_v \cap s'_v$  and  $\beta'_v \cap s_v$ ). Let  $\gamma = \sqrt{b/n}$ . We bound the number of vertices  $v$  with  $\alpha_v > \gamma$  differently from those with  $\alpha_v \leq \gamma$ .

Observe that the sum of the (turning) angles of the vertices of a single hole equals  $2\pi$ . Since the number of holes in  $I$  is  $O(b^2)$ , the sum of the angles of all union vertices in  $I$  is  $O(b^2)$  as well. The number of union vertices  $v$  with  $\alpha_v > \gamma$  inside  $I$  is therefore  $O(b^2/\gamma) = O(b\sqrt{bn})$ .

Observe that the length of the part inside  $I$  of each bounding line of a strip is at most  $D$ . Since the number of strips intersecting  $I$  is  $O(n)$ , the total length of all their bounding lines is  $O(Dn)$ . The intersection of the strips  $s_v$  and  $s'_v$  causes the interior of the segment  $\beta_v \cap s'_v$  adjacent to  $v$  to be in the interior of the union. For a vertex  $v$  with  $\alpha_v \leq \gamma$  this means that a stretch of length  $|\beta_v \cap s'_v| = \varepsilon / \sin \alpha_v \geq \varepsilon / \alpha_v \geq \varepsilon / \gamma$  on  $\beta_v$  adjacent to  $v$  cannot contain another union vertex. The number of union vertices  $v$  with  $\alpha_v > \gamma$  inside  $I$  is therefore  $O(nD/(\varepsilon/\gamma)) = O(b\sqrt{bn})$ .

### 3.2 Algorithms

As we observed, an approximate union of lines need only be a set of  $O(b^2)$  points (if  $b = O(n)$ ) that is a subset of a regular square grid with spacing  $\varepsilon$  inside  $I$ . For every point in this grid we need to find out whether it lies inside at least one of the  $n$  strips with width  $2\varepsilon$ , centered at the input lines. Every grid point inside some strip is guaranteed to be within  $\varepsilon$  of a line, and every point on an input line is guaranteed to be within  $\varepsilon/\sqrt{2}$  from a covered grid point. We can get the covered grid points in several ways. The easiest is to iterate over the strips, and for each strip find the  $O(b)$  points inside. This takes  $O(nb)$  time in total. In case we want to find a superset of the union of lines which is also an approximate union, we can take groups of four grid points covered by the same strip and put the whole grid square in the approximate union. For groups of three we take the triangle that is half a grid square.

A more efficient algorithm when  $b$  is sufficiently smaller than  $n$  is obtained by building a data structure on the strips, such that for a query point, we find out efficiently whether it lies inside some strip. Using partition trees, we can make an  $O(m)$  size data structure in  $O(m^{1+\delta})$  time such that queries can be answered in  $O(n^{1+\delta}/\sqrt{m})$  time, for any  $n \leq m \leq n^2$  [1]. We will query with the  $O(b^2)$  grid points, so we choose  $m$  to balance the preprocessing time and the total query time. The resulting algorithm takes  $O(n + b^2 + n^{2/3+\delta}b^{4/3})$  time.

**Theorem 1.** *An  $\varepsilon$ -approximate union of  $n$  lines inside a square of diameter  $D$  has complexity  $O(b^2)$  and can be computed in  $O(nb)$  and in  $O(n + n^{2/3+\delta}b^{4/3})$  time, for any  $\delta > 0$ , and where  $b = D/\varepsilon = O(n)$ .*

The union of strips is one of the problems known to be 3SUM-hard [10], implying that a subquadratic time algorithm for the construction is not likely to exist. But since we have  $\varepsilon$ -thick strips and are interested in a region of diameter  $D$  only, we can obtain subquadratic construction time.

To compute the union of  $n$   $\varepsilon$ -thick strips we will make use of multi-level partition trees and ray shooting [1]. All edges that appear in the thick union are segments on the set  $L$  of  $2n$  lines bounding the strips, and we will shoot along these lines to find the exposed edges. We build two data structures: one for querying when we are outside the union of the strips (to find the end of an exposed edge), and one for querying when we are inside the union of the strips (to find the start of an exposed edge). The former structure is easy. We simply preprocess all lines  $L$  bounding strips into a ray shooting structure.

It is more difficult to find the start of an exposed edge. When the starting point of the query ray is inside the union of the strips, we do not want to find the next intersection with some strip boundary, because it may not be part of the output, and we could be shooting  $\Theta(n^2)$  rays. Instead, we build a partition tree  $T$  of which the main tree allows us to select all strips that contain the query point  $q$  (start of the ray) represented by a small number of canonical nodes. For every node in the main tree we build an associated structure for ray shooting in the represented strips, *but from infinity and in the opposite direction*. This way we find the point  $q'$  where the ray will exit all strips in which the query point  $q$

lies. Either  $q'$  is a vertex of the union of strips and we start to find an exposed edge, or  $q'$  lies inside one or more strips, which necessarily are different from the ones that contained  $q$ . We use  $T$  to distinguish these cases. If  $q'$  lies inside, we continue to find the last exit point of the strips that contain  $q'$ , as we did before for  $q$ . It is clear that this approach will find the next starting point along the line we are processing.

**Lemma 5.** *Let  $q$  be any point inside the union of strips. If after at most two queries as specified above, the point  $q''$  lies in the union of strips, it has distance  $\geq \varepsilon$  from  $q$ .*

*Proof.* Suppose we query with  $q$  and find that  $q'$  is in the union. Then  $q'$  must lie in some strip  $S$  in which  $q$  does not lie. Point  $q''$  must lie such that  $\overline{qq''}$  cuts through  $S$  completely, which proves the lemma.

The lemma shows that the number of ray shooting queries inside the union is at most  $O(b)$  for any line, so  $O(nb)$  for all lines of  $L$  together. We also do at most  $O(n + b\sqrt{bn})$  queries outside the union due to the complexity bound. Since  $b = O(n)$ , we do  $O(nb)$  ray shootings. For both structures we have  $O(m^{1+\delta})$  preprocessing time and  $O(n^{1+\delta}/\sqrt{m})$  query time by the standard theory of partition trees [1]. We balance the preprocessing and the total query time and obtain an  $O(n^{4/3+\delta}b^{2/3})$  time bound to determine all exposed edges. It is easy to assemble these into the union of strips without affecting the time bound asymptotically.

**Theorem 2.** *The union of a set of  $n$   $\varepsilon$ -thick strips inside a square of diameter  $D$  has complexity  $O(n + b\sqrt{bn})$  can be constructed in  $O(n^{4/3+\delta}b^{2/3})$  time, for any  $\delta > 0$ , and where  $b = D/\varepsilon = O(n)$ .*

## 4 Minkowski Sums

Let  $P$  and  $Q$  be polygons with  $n$  vertices. We study the combinatorial complexity of approximate and thick Minkowski sums  $P \oplus Q$ . In addition we present algorithms for computing approximate and thick Minkowski sums.

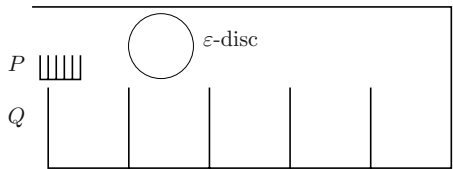
### 4.1 Complexity

The complexity of the Minkowski sum of two polygons with  $n$  vertices is  $O(n^4)$  and this bound is tight in the worst case. The following lemma says that approximations of lower complexity exist when the relative precision  $b = o(n^2)$ . We omit the proof as it is similar to that of Lemma 2.

**Lemma 6.** *Any Minkowski sum of two polygons with  $n$  vertices admits an  $\varepsilon$ -approximation of complexity  $O(\min(b^2, n^4))$ , and this bound is tight.*

We turn our attention to the thick Minkowski sum.

**Lemma 7.** *The complexity of the  $\varepsilon$ -thick Minkowski sum of two polygons with  $n$  vertices can be  $\Omega(n^2 + b^2)$ .*



**Fig. 2.** The Minkowski sum of  $P$ ,  $Q$  and an  $\varepsilon$ -disc can have  $\Omega(n^2)$  holes.

*Proof.* The bound of  $\Omega(b^2)$  is trivial. Figure 2 shows that  $P \oplus Q \oplus \Delta_\varepsilon$  can have  $\Omega(n^2)$  holes. The sum of the lengths of the  $\Theta(n)$  spikes of  $P$  and of  $Q$  is such that the distance from the  $\Theta(n^2)$  resulting spikes of  $P \oplus Q$  to the horizontal top bar of  $P \oplus Q$  is (just below)  $2\varepsilon$ . The Minkowski sum of  $P \oplus Q$  with  $\Delta_\varepsilon$  will have one tiny hole between each of the  $\Theta(n^2)$  pairs of supporting lines of consecutive spikes of  $P \oplus Q$ .

We deduce an upper bound on the complexity of the thick Minkowski sum by considering the complexity of the union boundary of all Minkowski sums of an edge of  $P$ , a vertex of  $Q$ , and  $\Delta_\varepsilon$ , and all Minkowski sums of an edge of  $Q$ , a vertex of  $P$ , and  $\Delta_\varepsilon$ . The boundary of the union of these racetracks (which are in fact thick translated edges of  $P$  or  $Q$ ) is a superset of the boundary of  $P \oplus Q \oplus \Delta_\varepsilon$ .

Let  $C$  be the set of  $O(n^2)$  racetracks. A racetrack is short if its diameter is at most  $4\varepsilon$  and long otherwise. Let  $C_s$  and  $C_l$  be the subsets of short and long racetracks in  $C$  respectively. We regard each racetrack as the union of two discs of diameter  $2\varepsilon$  and a rectangle of width  $2\varepsilon$ . Let  $S$  be the set of all discs resulting from racetracks in  $C$ , and let  $R_l$  be the set of rectangles—which are all longer than  $2\varepsilon$ —resulting from racetracks in  $C_l$ . We obtain a hexagonal inner approximation of a racetrack if we replace both discs by enclosed squares with side lengths  $\varepsilon\sqrt{2}$  such that the diagonals of these squares coincide with the sides of the rectangle that were originally covered by the discs. Observe that each hexagonal racetrack is the Minkowski sum of a translated copy  $e$  of an edge of  $P$  or  $Q$  with a square with side length  $\varepsilon\sqrt{2}$  whose diagonal is orthogonal to  $e$ . Let  $H_s$  be the set of all hexagonal racetracks resulting from racetracks in  $C_s$ .

It is well known that the union complexity of a collection of unit discs is linear in the number of discs. The complexity of  $\bigcup_{s \in S} s$ , and hence the number of vertices resulting from pairwise intersections of arcs in  $C$ , is therefore  $O(n^2)$ . For the sake of brevity, we let  $\bigcup X = \bigcup_{x \in X} x$  and  $\bigcup X, Y = (\bigcup_{x \in X} x) \cup (\bigcup_{y \in Y} y)$ . We derive a bound on the number of remaining vertices of the union  $\bigcup C = \bigcup C_s, C_l$  of racetracks in three steps. We will bound

- the number of vertices resulting from pairwise intersections of straight segments in  $C$  by considering the number of vertices in  $\bigcup R_l, H_s$ ,
- the number of vertices resulting from intersections of a circular arc from  $C$  and a straight segment from  $C_l$  by considering the number of vertices in  $\bigcup R_l, S$ , and

- the number of vertices resulting from intersections of a circular arc from  $C$  and a straight segment from  $C_s$  by considering the number of vertices in  $\bigcup C_s, S$ .

**Lemma 8.** *The number of holes in the union  $\bigcup R_l, H_s$  is  $O(n^2 + b^2)$ .*

*Proof.* The arrangement of line segments defined by the  $O(n^2)$  Minkowski sums of an edge of  $P$  or  $Q$  and a vertex of  $Q$  or  $P$  has  $O(b^2)$  convex holes of area at least  $\Omega(\varepsilon^2)$ , and  $O(n^2)$  non-convex holes, as each of these holes must contain an endpoint of a segment. Replacing the line segments by the corresponding rectangles or hexagonal racetracks may lead to splits of holes. If one of the six vertices of a racetrack  $h \in H_s$  splits a hole, then that particular vertex ends up inside  $\bigcup R_l, H_s$ . If an edge of  $h$  splits a hole, then a vertex of another racetrack or rectangle ends up inside  $h$  and thus inside  $\bigcup R_l, H_s$ . Similar arguments apply to the four vertices and edges of a rectangle  $r \in R_l$ . As a result, the number of newly created holes is  $O(n^2)$ .

The bound on the number of holes plays a role in the following result, whose proof is given in the appendix.

**Lemma 9.** *The complexity of the union  $\bigcup R_l$  is  $O((n + b)n\sqrt{b \log n} + n^2 \log n)$ .*

Addition of the short hexagonal racetracks of  $H_s$  does not increase the asymptotic complexity of the union.

**Corollary 1.** *The complexity of the union  $\bigcup R_l, H_s$  is  $O((n + b)n\sqrt{b \log n} + n^2 \log n)$ .*

The following interesting theorem combines the union complexity of discs and rectangles. It is a variation on a result by Pach and Sharir [17]. We emphasize that a unit-width rectangle is a rectangle whose shortest side has unit length. The theorem provides the key to our bound for the complexity of  $\bigcup R_l, S$ .

**Theorem 3.** *Given a set  $R$  of  $m$  unit-width rectangles with union complexity  $O(f(m))$  and a set  $S$  of  $n$  unit-diameter discs. Then the complexity of union of the rectangles from  $R$  and the discs from  $S$  is  $O(f(m) + n)$ .*

There are simple examples that show that the theorem is not true if discs can be arbitrarily large, or rectangles arbitrarily short. In both cases an  $\Theta(nm)$  complexity construction can be made. These examples and a proof of the theorem are given in the full paper.

We now apply the result of Lemma 9 in Theorem 3 and obtain a complexity bound on the union of the rectangles in  $R_l$  and the discs in  $S$ , which leads to the following result.

**Corollary 2.** *The complexity of the union  $\bigcup R_l, S$  is  $O((n + b)n\sqrt{b \log n} + n^2 \log n)$ .*



It remains to bound the complexity of  $\bigcup C_s, S$ . The shapes in  $C_s \cup S$  are all fat and have comparable sizes. Let  $\tau$  be an equilateral triangle whose sides have length  $\varepsilon\sqrt{3}$ . For each shape  $f \in C_s \cup S$  and each point  $p \in \partial f$ , we can place  $\tau$  fully inside  $f$  while one of its corners coincides with  $p$ . Moreover, we observe that the diameter of any  $f \in C_s \cup S$  is between  $2\varepsilon$  and  $4\varepsilon$ . Efrat [5] states that in this case the union complexity is  $O(\lambda_{s+2}(|C_s \cup S|))$ , where  $s$  is the maximum number of boundary intersections of any two shapes from  $C_s \cup S$ .

**Lemma 10.** *The complexity of the unions  $\bigcup C_s, S$  is  $O(\lambda_6(n^2))$ .*

The bounds on the complexities of the unions  $\bigcup R_l, H_s, \bigcup R_l, S$ , and  $\bigcup C_s, S$  imply the following bound for the  $\varepsilon$ -thick Minkowski sum.

**Lemma 11.** *The complexity of the  $\varepsilon$ -thick Minkowski sum of two polygons with  $n$  vertices is  $O((n+b)n\sqrt{b}\log n + n^2\log n)$ .*

## 4.2 Algorithms

The approximate Minkowski sum of two simple polygons can be computed in the same way as for approximate union of lines. We determine for each of the  $O(b^2)$  points of an appropriately defined grid whether it is inside or outside the Minkowski sum. There are several ways to do this.

If  $b$  is small, we can test each grid point without preprocessing in  $O(n)$  time. The test whether a given point—which we can assume to be the origin after a translation—lies in the Minkowski sum of two polygons  $P$  and  $Q$  reduces to the intersection test for two simple polygons  $P$  and  $-Q$  (which is  $Q$  mirrored in the origin). This on its turn can be solved by testing if a (combined) polygon has self-intersections: compute a line segment  $s$  that connects the boundaries of  $P$  and  $-Q$  without intersecting  $P$  and  $-Q$  elsewhere, and convert  $s$  into a narrow passage to make a polygon that is a combination  $\Gamma_s(P, Q)$  of  $P$  and  $-Q$ . The test whether polygon  $\Gamma_s(P, Q)$  is simple determines whether  $P$  and  $Q$  intersect, and therefore whether the origin lies in the Minkowski sum of  $P$  and  $Q$ . The simplicity test takes linear time as a by-product of Chazelle's linear time triangulation algorithm of a simple polygon [2]. Concluding, we can compute an approximate Minkowski sum of two simple polygons with  $n$  vertices in total in  $O(nb^2)$  time.

For larger values of  $b$  it becomes more efficient to see the Minkowski sum of  $P$  and  $Q$  as the union of  $O(n^2)$  triangles. We can store them in a partition tree such that point containment queries can be answered efficiently [1]. Using the preprocessing versus total query time trade-off for range searching we obtain an  $O(n^2 + b^2 + (nb)^{4/3+\delta})$  time algorithm for any constant  $\delta > 0$ . In the same way as for the approximate union of lines we can make the approximate Minkowski sum a connected subset of the plane.

**Theorem 4.** *An  $\varepsilon$ -approximate Minkowski sum of two simple polygons with  $n$  vertices in total, inside a square of diameter  $D$  has complexity  $O(b^2)$  and can be computed in  $O(nb^2)$  and in  $O(n^2 + b^2 + (nb)^{4/3+\delta})$  time, for any  $\delta > 0$ , and where  $b = D/\varepsilon = O(n^2)$ .*

To compute the thick Minkowski sum of two simple polygons  $P$  and  $Q$  we use the ray shooting approach of Section 3. However, we cannot do exactly the same, because a ray that enters the union of rectangles and discs is not guaranteed to stay inside the union over a length at least  $\varepsilon$ . Furthermore, we have to deal with the circular parts of the thick Minkowski sum.

For convenience we will compute the Minkowski sum of every vertex of  $P$ , edge of  $Q$ , and  $\Delta_\varepsilon$ , and every vertex of  $Q$ , edge of  $P$ , and  $\Delta_\varepsilon$ . The union of these  $O(n^2)$  racetracks is a subset of the thick Minkowski sum. It has the same asymptotic complexity, and from it the true thick Minkowski sum can be constructed easily by filling holes and removing boundary edges of these holes.

Let  $C$  again be the set of  $O(n^2)$  racetracks. We consider each racetrack to be the union of two discs and a rectangle as before, giving a set  $S$  of  $O(n^2)$  discs and a set  $R$  of  $O(n^2)$  rectangles. The union of the discs has complexity  $O(n^2)$  and can be constructed in  $O(n^2 \log^2 n)$  time by the algorithm of Kedem et al. [12]. The union of the rectangles has complexity  $O((n+b)n\sqrt{b}\log n + n^2 \log n)$ , and we show next how to compute it. Recall that every rectangle has width  $2\varepsilon$ , and the length can vary from nearly 0 to linear in  $D$ . Let  $R_s$  be the subset of (short) rectangles of length at most  $4\varepsilon$  and let  $R_l$  be the rectangles of length  $> 4\varepsilon$ . For each rectangle in  $R_l$ , we cut off two squares of size  $2\varepsilon$  by  $2\varepsilon$ , one from either end. Let  $R'_l$  be the set of shortened rectangles from  $R_l$ , and let  $R'_s$  be the union of  $R_s$  and the  $2|R_l|$  squares that were cut off.

We preprocess  $R'_s$  as follows. Compute the union of the rectangles in  $R'_s$  (through their hexagonal racetrack) explicitly with the algorithm of Kedem et al. in  $O(\lambda_{14}(n^2) \log^2 n)$  time. It has complexity  $O(\lambda_{14}(n^2))$  by Efrat's result [5]. (Two hexagons can intersect 12 times). Preprocess the edges of the boundary of the union into a ray shooting structure. The start of the query ray can be inside or outside the union.

We preprocess  $R'_l$  as in Section 3: the main tree is a partition tree in which all rectangles that contain a query point (start of the query ray) can be selected in a small number of canonical subsets, represented by nodes of the tree. For every node of the tree we store an associated structure for ray shooting 'from the other side', so we can find the last exit of the rectangles of  $R'_l$ .

A query is always performed in both structures. There are four cases for the start  $q$  of the query ray:

1. If  $q \notin \bigcup R'_s$  and  $q \notin \bigcup R'_l$ , then the (double) ray shooting query will give the next vertex along the ray, which is in the union  $\bigcup R$  as well.
2. If  $q \in \bigcup R'_s$  and  $q \notin \bigcup R'_l$ , then either the query ray exits  $\bigcup R'_s$  first, in which case we find a vertex of the union, or the query ray enters  $\bigcup R'_l$  first. In the latter case, the next ray shooting query is done from the exit point of  $\bigcup R'_s$ .
3. If  $q \notin \bigcup R'_s$  and  $q \in \bigcup R'_l$ , then either the query ray exits the subset of the rectangles of  $R'_l$  that contain it first, or the query ray enters  $\bigcup R'_s$  first. In the latter case, the next ray shooting query can start at the exit of the subset from  $\bigcup R'_l$ . In the former case, we either exit  $\bigcup R'_l$  and find a vertex of the union, or we stay inside  $\bigcup R'_l$  and the next query will be of type 3 again.

4. If  $q \in \bigcup R'_s$  and  $q \in \bigcup R'_l$ , then either the query ray exits  $\bigcup R'_s$  last, or the query ray exits the subset of the rectangles of  $R'_l$  that contain it last. We will shoot from the furthest point hit, and the query can be of types 1, 2, 3, or 4.

We will perform the ray shooting queries along the sides of the rectangles in  $R$ , so we determine  $\bigcup R$  this way. We can prove that ray shooting queries find a vertex of the union, or the starting point of a constant number of queries later, the query ray advances in a way similar to Lemma 5.

**Lemma 12.** *If five consecutive ray shooting queries stay fully inside  $\bigcup R$ , then the distance from the start of the first and end of the fifth ray is at least  $2\varepsilon$ .*

We showed that the total number of queries done is bounded by  $O(n^2b)$  plus the union complexity of  $\bigcup R$ . Since we assume  $b = O(n^2)$ , we do  $O(n^2b + (n + b)n\sqrt{b\log n} + n^2\log n)$  queries, and after balancing the preprocessing time and total query time we spend  $O(n^{8/3+\delta}b^{2/3})$  time on ray shooting to compute  $\bigcup R$ .

After computing the union  $\bigcup R$  of rectangles and the union  $\bigcup S$  of discs we compute the union of unions with a simple overlay, for example by a plane sweep. Every new intersection point appears in the thick Minkowski sum, so the overlay step takes  $O((n^2\sqrt{b} + nb\sqrt{b} + \lambda_{14}(n^2))\log^{2.5} n + n^2\log^3 n)$  time. This is always dominated by the total ray shooting time.

**Theorem 5.** *The  $\varepsilon$ -thick Minkowski sum of two simple polygons with  $n$  vertices in total inside a square of diameter  $D$  has complexity  $O((n + b)n\sqrt{b\log n} + n^2\log n)$  and can be constructed in  $O(n^{8/3+\delta}b^{2/3})$  time, for any  $\delta > 0$ , and where  $b = D/\varepsilon = O(n^2)$ .*

## 5 Conclusions and Open Problems

We have studied the fundamental geometric concept of Minkowski sums in a context that bears resemblance to the study of realistic input models. We have presented new combinatorial and algorithmic results on approximate and thick unions of lines, and approximate and thick Minkowski sums. The results are related to fatness and realistic input models, but we take the fatness parameter into account explicitly. Although we studied unions of lines and Minkowski sums, our methods can be used for approximate and thick unions of triangles as well. For  $\varepsilon$ -approximate unions of  $n$  triangles in a region of interest with diameter  $D$ , the complexity is  $\Theta(b^2)$  in the worst case, and we can compute it in  $O(n + n^{2/3+\delta}b^{4/3})$  time, where  $\delta > 0$  and  $b = D/\varepsilon = O(n)$ . For thick unions of triangles, we get an  $\Omega(b^2 + n)$  and  $O(n\log n + b\sqrt{bn\log n})$  complexity bound and an  $O(n^{4/3+\delta}b^{2/3})$  time algorithm.

The main open problems are closing the gaps left in the complexity bounds and improving the running time of the algorithms. We believe that in any case the  $\sqrt{\log n}$  factor in the upper bound for thick Minkowski sums is an artifact of the proof. It would also be interesting to extend the combination theorem for unions of unit-width rectangles and unit-width discs to the case where the radius of the discs may be smaller.

## References

1. P.K. Agarwal. Range searching. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 36, pages 809–837. CRC Press, Boca Raton, 2nd edition, 2004.
2. B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6(5):485–524, 1991.
3. K. Daniels and V.J. Milenkovic. Multiple translational containment, part i: An approximate algorithm. *Algorithmica*, 19(1–2):148–182, September 1997.
4. M. de Berg, M.J. Katz, A.F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. *Algorithmica*, 34:81–97, 2002.
5. A. Efrat. The complexity of the union of  $(\alpha, \beta)$ -covered objects. In *Proc. 15th Annu. ACM Sympos. Computat. Geometry*, 1999.
6. A. Efrat, M.J. Katz, F. Nielsen, and M. Sharir. Dynamic data structures for fat objects and their applications. *Comput. Geom. Theory Appl.*, 15:215–227, 2000.
7. A. Efrat, G. Rote, and M. Sharir. On the union of fat wedges and separating a collection of segments by a line. *Comput. Geom. Theory Appl.*, 3:277–288, 1993.
8. A. Efrat and M. Sharir. On the complexity of the union of fat objects in the plane. *Discrete Comput. Geom.*, 23:171–189, 2000.
9. J. Erickson. Local polyhedra and geometric graphs. In *Proc. 19th Annu. ACM Sympos. Computat. Geometry*, pages 171–180, 2003.
10. A. Gajentaan and M.H. Overmars. On a class of  $O(n^2)$  problems in computational geometry. *Comput. Geom. Theory Appl.*, 5:165–185, 1995.
11. R. Heckmann and T. Lengauer. Computing upper and lower bounds on textile nesting problems. In *Proc. 4th Annu. European Sympos. Algorithms*, volume 1136 of *Lecture Notes Comput. Sci.*, pages 392–405. Springer-Verlag, 1996.
12. K. Kedem, R. Livne, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete Comput. Geom.*, 1:59–71, 1986.
13. J.-C. Latombe. *Robot Motion Planning*. Kluwer Ac. Pub. Boston, 1991.
14. J. Matoušek, J. Pach, M. Sharir, S. Sifrony, and E. Welzl. Fat triangles determine linearly many holes. *SIAM J. Comput.*, 23:154–169, 1994.
15. V.J. Milenkovic. Multiple translational containment, part ii: Exact algorithm. *Algorithmica*, 19(1–2):183–218, September 1997.
16. M.H. Overmars and A.F. van der Stappen. Range searching and point location among fat objects. *Journal of Algorithms*, 21:629–656, 1996.
17. J. Pach and M. Sharir. On the boundary of the union of planar convex sets. *Discrete Comput. Geom.*, 21:321–328, 1999.
18. J. Pach and G. Tardos. On the boundary complexity of the union of fat triangles. *SIAM J. Comput.*, 31:1745–1760, 2002.
19. M. Sharir. Algorithmic motion planning. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 47, pages 1037–1064. CRC Press, Boca Raton, 2nd edition, 2004.
20. A.F. van der Stappen and M.H. Overmars. Motion planning amidst fat obstacles. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 31–40, 1994.
21. M. van Kreveld and B. Speckmann. Cutting a country for smallest square fit. In *Proc. ISAAC'02*, volume 2518 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 2002.
22. M. van Kreveld. On fat partitioning, fat covering, and the union size of polygons. *Comput. Geom. Theory Appl.*, 9(4):197–210, 1998.

# Radio Network Clustering from Scratch

Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer

Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland

{kuhn,moscitho,wattenhofer}@inf.ethz.ch

**Abstract.** We propose a novel randomized algorithm for computing a dominating set based clustering in wireless ad-hoc and sensor networks. The algorithm works under a model which captures the characteristics of the set-up phase of such multi-hop radio networks: asynchronous wake-up, the hidden terminal problem, and scarce knowledge about the topology of the network graph. When modelling the network as a unit disk graph, the algorithm computes a dominating set in polylogarithmic time and achieves a constant approximation ratio.

## 1 Introduction

Ad-hoc and sensor networks are formed by autonomous nodes communicating via radio, without any additional infrastructure. In other words, the communication “infrastructure” is provided by the nodes themselves. When being deployed, the nodes initially form an *unstructured* radio network, which means that no reliable and efficient communication pattern has been established yet. Before any reasonable communication can be carried out, the nodes must establish a media access control (MAC) scheme which provides reliable point-to-point connections to higher-layer protocols and applications. The problem of setting up an initial structure in radio networks is of great importance in practice. Even in a single-hop ad-hoc network such as Bluetooth and for a small number of devices, the initialization tends to be slow. Clearly, in a multi-hop scenario with many nodes, the time consumption for establishing a communication pattern increases even further. In this paper, we address this *initialization process*.

One prominent approach to solving the problem of bringing structure into a multi-hop radio network is a *clustering*, in which each node in the network is either a cluster-head or has a cluster-head within its communication range (such that cluster-heads can act as coordination points for the MAC scheme) [1,4,6]. When we model a multi-hop radio network as a graph  $G = (V, E)$ , this clustering can be formulated as a classic graph theory problem: In a graph, a *dominating set* is a subset  $S \subseteq V$  of nodes such that for every node  $v$ , either a)  $v \in S$  or b)  $v' \in S$  for a direct neighbor  $v'$  of  $v$ . As it is desirable to compute a dominating set with few *dominators*, we study the *minimum dominating set* (MDS) problem which asks for a dominating set of minimum cardinality.

The computation of dominating sets for the purpose of structuring networks has been studied extensively and a variety of algorithms have been proposed, e.g. [4,7,9,10,12]. To the best of our knowledge, all these algorithms operate on an existing MAC layer, providing point-to-point connections between neighboring nodes. While this is valid in structured networks, it is certainly an improper assumption for the initialization phase. In fact, by assuming point-to-point connections, many vital problems arising in unstructured

networks (collision detection, asynchronous wake-up, or the hidden terminal problem) are simply abstracted away. Consequently, none of the existing dominating set algorithms helps in the initialization process of such networks.

We are interested in a simple and practical algorithm which quickly computes a clustering from *scratch*. Based on this initial clustering, the MAC layer can subsequently be established. An unstructured multi-hop radio network can be modelled as follows:

- The network is *multi-hop*, that is, there exist nodes that are not within their mutual transmission range. Being multi-hop complicates things since some of the neighbors of a sending node may receive a transmission, while others are experiencing interference from other senders and do not receive the transmission.
- The nodes do not feature a reliable *collision detection* mechanism [2,5,8]. In many scenarios not assuming any collision detection mechanism is realistic. Nodes may be tiny sensors in a sensor network where equipment is restricted to the minimum due to limitations in energy consumption, weight, or cost. The absence of collision detection includes sending nodes, i.e., the sender does not know whether its transmission was received successfully or whether it caused a collision.
- Our model allows nodes to wake-up *asynchronously*. In a multi-hop environment, it is realistic to assume that some nodes wake up (e.g. become deployed, or switched on) later than others. Consequently, nodes do not have access to a global clock. Asynchronous wake-up rules out an ALOHA-like MAC schemes as this would result in a linear runtime in case only one single node wakes up for a long time.
- Nodes have only limited knowledge about the total number of nodes in the network and no knowledge about the nodes' distribution or wake-up pattern.

In this paper, we present a randomized algorithm which computes an asymptotically optimal clustering for this harsh model in polylogarithmic time only. Section 2 gives an overview over relevant previous work. Section 3 introduces our model as well as some well-known facts. The algorithm is developed and analyzed in Sections 4 and 5.

## 2 Related Work

The problem of finding a minimum dominating set was proven to be NP-hard. Furthermore, it has been shown in [3] that the best possible approximation ratio for this problem is  $\ln \Delta$  where  $\Delta$  is the highest degree in the graph, unless NP has deterministic  $n^{O(\log \log n)}$ -time algorithms. For unit disk graphs, the problem remains NP-hard, but constant factor approximations are possible. Several distributed algorithms have been proposed, both for general graphs [7,9,10] and the Unit Disk Graph [4,12]. All the above algorithms assume point-to-point connections between neighboring nodes and are thus unsuitable in the context of initializing radio networks.

A model similar to the one used in this paper has previously been studied in the context of analyzing the complexity of broadcasting in multi-hop radio networks, e.g. [2]. A striking difference to our model is that throughout the literature on broadcast in radio networks, *synchronous wake-up* is considered, i.e. all nodes have access to a global clock and start the algorithm simultaneously. A model featuring asynchronous wake-up has been studied in recent papers on the *wake-up problem* in single-hop networks [5,8].

In comparison to our model, these papers define a much *weaker notion of asynchrony*. Particularly, it is assumed that sleeping nodes are *woken up* by a successfully transmitted message. In a single-hop network, the problem of waking up all nodes thus reduces to analyzing the number of time-slots until one message is successfully transmitted. While this definition of asynchrony leads to theoretically interesting problems and algorithms, it does not closely reflect reality.

### 3 Model

We model the *multi-hop* radio network with the well known *Unit Disk Graph* (UDG). In a UDG  $G = (V, E)$ , there is an edge  $\{u, v\} \in E$  iff the Euclidean distance between  $u$  and  $v$  is at most 1. Nodes may wake up *asynchronously* at any time. We call a node *sleeping* before its wake-up, and *active* thereafter. Sleeping nodes can neither send nor receive any messages, regardless of their being within the transmission range of a sending node. Nodes do not have any a-priori knowledge about the topology of the network. They only have an upper bound  $\hat{n}$  on the number of nodes  $n = |V|$  in the graph. While  $n$  is unknown, all nodes have the same estimate  $\hat{n}$ . As shown in [8], without any estimate of  $n$  and in absence of a global clock, every algorithm requires at least time  $\Omega(n/\log n)$  until one single message can be transmitted without collision.

While our algorithm does not rely on synchronized time-slots in any way, we do assume time to be divided into time-slots in the analysis section. This simplification is justified due to the trick used in the analysis of slotted vs. unslotted ALOHA [11], i.e., a single packet can cause interference in no more than two consecutive time-slots. Thus, an analysis in an “ideal” setting with synchronized time-slots yields a result which is only by a constant factor better as compared to the more realistic unslotted setting.

We assume that nodes have three independent communication channels  $\Gamma_1$ ,  $\Gamma_2$ , and  $\Gamma_3$  which may be realized with an FDMA scheme. In each time-slot, a node can either send or not send. Nodes do not have a *collision detection mechanism*, that is, nodes are unable to distinguish between the situation in which two or more neighbors are sending and the situation in which no neighbor is sending. A node receives a message on channel  $\Gamma$  in a time-slot only if *exactly one neighbor* has sent a message in this time-slot on  $\Gamma$ . A sending node does not know how many (if any at all!) neighbors have correctly received its transmission. The variables  $p_k$  and  $q_k$  denote the probabilities that node  $k$  sends a message in a given time-slot on  $\Gamma_1$  and  $\Gamma_2$ , respectively. Unless otherwise stated, we use the term *sum of sending probabilities* to refer to the sum of sending probabilities on  $\Gamma_1$ . We conclude this section with two facts. The first was proven in [8] and the second can be found in standard mathematical textbooks.

**Fact 1.** *Given a set of probabilities  $p_1 \dots p_n$  with  $\forall i : p_i \in [0, \frac{1}{2}]$ , the following inequalities hold:  $(1/4)^{\sum_{k=1}^n p_k} \leq \prod_{k=1}^n (1 - p_k) \leq (1/e)^{\sum_{k=1}^n p_k}$ .*

**Fact 2.** *For all  $n, t$ , with  $n \geq 1$  and  $|t| \leq n$ ,  $e^t (1 - t^2/n) \leq (1 + t/n)^n \leq e^t$ .*



**Algorithm 1** Dominator Algorithm

---

```

decided := dominator := false;
upon wake-up do
1: for  $j := 1$  to  $\delta \cdot \lceil \log \hat{n} \rceil$  by 1 do
2:   if message received in current time-slot then decided := true; fi
3: end for
4: for  $j := \lceil \log \hat{n} \rceil$  to 0 by -1 do
5:    $p := 1 / (2^{j+\beta})$ ;
6:   for  $i := 1$  to  $\delta$  by 1 do
7:      $b_i^{(1)} := 0$ ;  $b_i^{(2)} := 0$ ;  $b_i^{(3)} := 0$ ;
8:     if not decided then
9:        $b_i^{(1)} := 1$ , with probability  $p$ ;
10:      if  $b_i^{(1)} = 1$  then dominator := true;
11:      else if message received in current time-slot then decided := true;
12:      fi
13:    end if
14:    if dominator then
15:       $b_i^{(2)} := 1$ , with probability  $q$ ;  $b_i^{(3)} := 1$ , with probability  $q / \log \hat{n}$ ;
16:    end if
17:    if  $b_i^{(1)} = 1$  then send message on  $\Gamma_1$  fi
18:    if  $b_i^{(2)} = 1$  then send message on  $\Gamma_2$  fi
19:    if  $b_i^{(3)} = 1$  then send message on  $\Gamma_3$  fi
20:  end for
21: end for
22: if not decided then dominator := decided := true; fi
23: if dominator then
24:   loop
25:    send message on  $\Gamma_2$  and  $\Gamma_3$ , with probability  $q$  and  $q / \log \hat{n}$ , respectively;
26:   end loop
27: end if

```

---

## 4 Algorithm

A node starts executing the dominator algorithm (Algorithm 1) upon waking up. In the first phase (lines 1 to 3), nodes wait for messages (on all channels) without sending themselves. The reason is that nodes waking up late should not interfere with already existing dominators. Thus, a node first listens for existing dominators in its neighborhood before actively trying to become dominator itself.

The main part of the algorithm (starting in line 4) works in rounds, each of which contains  $\delta$  time-slots. In every time-slot, a node sends with probability  $p$  on channel  $\Gamma_1$ . Starting from a very small value, this sending probability  $p$  is doubled (lines 4 and 5) in every round. When sending its first message, a node becomes a dominator and, in addition to its sending on channel  $\Gamma_1$ , it starts sending on channels  $\Gamma_2$  and  $\Gamma_3$  with probability  $q$  and  $q / \log n$ , respectively. Once a node becomes a dominator, it will remain so for the rest of the algorithm's execution. For the algorithm to work properly, we must prevent the sum of sending probabilities on channel  $\Gamma_1$  from reaching too high values. Otherwise,



too many collisions will occur, leading to a large number of dominators. Hence, upon receiving its first message (without collision) on any channel, a node becomes *decided* and stops sending on  $\Gamma_1$ . Being decided means that the node is covered by a dominator and consequently, the node stops sending on  $\Gamma_1$ .

Thus, the basic intuition is that nodes, after some initial listening period, compete to become dominator by exponentially increasing their sending probability on  $\Gamma_1$ . Channels  $\Gamma_2$  and  $\Gamma_3$  then ensure that the number of further dominators emerging in the neighborhood of an already existing dominator is bounded.

The parameters  $q$ ,  $\beta$ , and  $\delta$  of the algorithm are chosen as to optimize the results and guarantee that all claims hold with high probability. In particular, we define  $q := (2^\beta \cdot \lceil \log \hat{n} \rceil)^{-1}$ ,  $\delta := \lceil \log(\hat{n}) / \log(503/502) \rceil$ , and  $\beta := 6$ . The parameter  $\delta$  is chosen large enough to ensure that with high probability, there is a round in which at least one competing node will send without collision. The parameter  $q$  is chosen such that during the “waiting time-slots”, a new node will receive a message from an existing dominator. Finally,  $\beta$  maximizes the probability of a successful execution of the algorithm. The algorithm’s correctness and time-complexity (defined as the number of time-slots of a node between wake-up and decision) follow immediately:

**Theorem 1.** *The algorithm computes a correct dominating set. Moreover, every node decides whether or not to become dominator in time  $O(\log^2 \hat{n})$ .*

*Proof.* The first for-loop is executed  $\delta \cdot \lceil \log \hat{n} \rceil$  times. The two nested loops of the algorithm are executed  $\lceil \log \hat{n} \rceil + 1$  and  $\delta$  times, respectively. After these two loops, all remaining undecided nodes decide to become dominator.

## 5 Analysis

In this section, we show that the expected number of dominators in the network is within  $O(1)$  of an optimal solution. As argued in Section 3, we can simplify the analysis by assuming all nodes operate in synchronized time-slots.

We cover the plane with circles  $C_i$  of radius  $r = 1/2$  by a hexagonal lattice shown

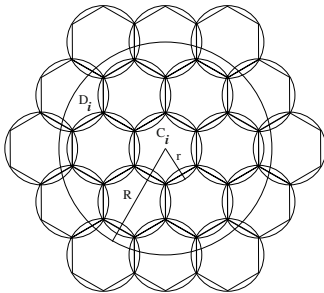


Fig. 1. Circles  $C_i$  and  $D_i$

in Figure 1. Let  $D_i$  be the circle centered at the center of  $C_i$  having radius  $R = 3/2$ . It can be seen in Figure 1 that  $D_i$  is (fully or partially) covering 19 smaller circles  $C_j$ . Note that every node in a circle  $C_i$  can hear all other nodes in  $C_i$ . Nodes outside  $D_i$  are not able to cause a collision in  $C_i$ .

The proof works as follows. We first bound the sum of sending probabilities in a circle  $C_i$ . This leads to an upper bound on the number of collisions in a circle

before at least one dominator emerges. Next, we give a probabilistic bound on the number of sending nodes per collision. In the last step, we show that nodes waking up being already covered do not become dominator. All these claims hold with high

probability. Note that for the analysis, it is sufficient to assume  $\hat{n} = n$ , because solving minimum dominating set for  $n' < n$  cannot be more difficult than for  $n$ . If it were, the imaginary adversary controlling the wake-ups of all nodes could simply decide to let  $n - n'$  sleep infinitely long, which is indistinguishable from having  $n'$  nodes.

**Definition 1.** Consider a circle  $C_i$ . Let  $t$  be a time-slot in which a message is sent by a node  $v \in C_i$  on  $\Gamma_1$  and received (without collision) by all other nodes in  $C_i$ . We say that circle  $C_i$  clears itself in time-slot  $t$ . Let  $t_0$  be the first such time-slot. We say that  $C_i$  terminates itself in time-slot  $t_0$ . For all time-slots  $t' \geq t_0$ , we call  $C_i$  terminated.

**Definition 2.** Let  $s(t) := \sum_{k \in C_i} p_k$  be the sum of sending probabilities on  $\Gamma_1$  in  $C_i$  at time  $t$ . We define the time slot  $t_i^j$  so that for the  $j^{\text{th}}$  time in  $C_i$ , we have  $s(t_i^j - 1) < \frac{1}{2^\beta}$  and  $s(t_i^j) \geq \frac{1}{2^\beta}$ . We further define the Interval  $\mathcal{I}_i^j := [t_i^j \dots t_i^j + \delta - 1]$ .

In other words,  $t_i^0$  is the time-slot in which the sum of sending probabilities in  $C_i$  exceeds  $\frac{1}{2^\beta}$  for the first time and  $t_i^j$  is the time-slot in which this threshold is surpassed for the  $j^{\text{th}}$  time in  $C_i$ .

**Lemma 1.** For all time-slots  $t' \in \mathcal{I}_i^j$ , the sum of sending probabilities in  $C_i$  is bounded by  $\sum_{k \in C_i} p_k \leq 3/2^\beta$ .

*Proof.* According to the definition of  $t_i^j$ , the sum of sending probabilities  $\sum_{k \in C_i} p_k$  at time  $t_i^j - 1$  is less than  $\frac{1}{2^\beta}$ . All nodes which are active at time  $t_i^j$  will double their sending probability  $p_k$  exactly once in the following  $\delta$  time-slots. Previously inactive nodes may wake up during that interval. There are at most  $n$  of such newly active nodes and each of them will send with the initial sending probability  $\frac{1}{2^\beta \hat{n}}$  in the given interval. In  $\mathcal{I}_i^j$ , we get

$$\sum_{k \in C_i} p_k \leq 2 \cdot \frac{1}{2^\beta} + \sum_{k \in C_i} \frac{1}{2^\beta \hat{n}} \leq 2 \cdot \frac{1}{2^\beta} + \frac{n}{2^\beta \hat{n}} \leq \frac{3}{2^\beta}.$$

Using the above lemma, we can formulate a probabilistic bound on the sum of sending probabilities in a circle  $C_i$ . Intuitively, we show that before the bound can be surpassed,  $C_i$  does either clear itself or some nodes in  $C_i$  become decided such that the sum of sending probabilities decreases.

**Lemma 2.** The sum of sending probabilities of nodes in a circle  $C_i$  is bounded by  $\sum_{k \in C_i} p_k \leq 3/2^\beta$  with probability at least  $1 - o(\frac{1}{n^2})$ . The bound holds for all  $C_i$  in  $G$  with probability at least  $1 - o(\frac{1}{n})$ .

*Proof.* The proof is by induction over all intervals  $\mathcal{I}_i^j$  and the corresponding time-slots  $t_i^j$  in ascending order. Lemma 1 states that the sum of sending probabilities in  $C_i$  is bounded by  $\frac{3}{2^\beta}$  in each interval  $\mathcal{I}_i^j$ . In the sequel, we show that in  $\mathcal{I}_i^j$ , the circle  $C_i$  either clears itself or the sum of sending probabilities falls back below  $\frac{1}{2^\beta}$  with high probability. Note that for all time-slots  $t$  not covered by any

interval  $\mathcal{I}_i^j$ , the sum of sending probabilities is below  $\frac{1}{2^\beta}$ . Hence, the induction over all intervals is sufficient to prove the claim.

Let  $t' := t_i^0$  be the very first critical time-slot in the network and let  $\mathcal{I}'$  the corresponding interval. If some of the active nodes in  $C_i$  receive a message from a neighboring node, the sum of sending probabilities may fall back below  $\frac{1}{2^\beta}$ . In this case, the sum does obviously not exceed  $\frac{3}{2^\beta}$ . If the sum of sending probabilities does not fall back below  $\frac{1}{2^\beta}$ , the following two inequalities hold for the duration of the interval  $\mathcal{I}'$ :

$$\frac{1}{2^\beta} \leq \sum_{k \in C_i} p_k \leq \frac{3}{2^\beta} : \text{in } C_i \quad (1)$$

$$0 \leq \sum_{k \in C_j} p_k \leq \frac{3}{2^\beta} : \text{in } C_j \in D_i, i \neq j. \quad (2)$$

The second inequality holds because  $t'$  is the very first time-slot in which the sum of sending probabilities exceeds  $\frac{1}{2^\beta}$ . Hence, in each  $C_j \in D_i$ , the sum of sending probabilities is at most  $\frac{3}{2^\beta}$  in  $\mathcal{I}'$ . (Otherwise, one of these circles would have reached  $\frac{1}{2^\beta}$  before circle  $C_i$  and  $t'$  is not the first time-slot considered).

We will now compute the probability that  $C_i$  clears itself within  $\mathcal{I}'$ . Circle  $C_i$  clears itself when exactly one node in  $C_i$  sends and no other node in  $D_i \setminus C_i$  sends. The probability  $P_0$  that no node in any neighboring circle  $C_j \in D_i, j \neq i$  sends is

$$\begin{aligned} P_0 &= \prod_{\substack{C_j \in D_i \\ j \neq i}} \prod_{k \in C_j} (1 - p_k) \stackrel{\text{Fact 1}}{\geq} \prod_{\substack{C_j \in D_i \\ j \neq i}} \left(\frac{1}{4}\right)^{\sum_{k \in C_j} p_k} \\ &\stackrel{\text{Lemma 1}}{\geq} \prod_{\substack{C_j \in D_i \\ j \neq i}} \left(\frac{1}{4}\right)^{\frac{3}{2^\beta}} \geq \left[\left(\frac{1}{4}\right)^{\frac{3}{2^\beta}}\right]^{18}. \end{aligned} \quad (3)$$

Let  $P_{suc}$  be the probability that exactly one node in  $C_i$  sends:

$$\begin{aligned} P_{suc} &= \sum_{k \in C_i} \left( p_k \cdot \prod_{\substack{l \in C_i \\ l \neq k}} (1 - p_l) \right) \geq \sum_{k \in C_i} p_k \cdot \prod_{l \in C_i} (1 - p_l) \\ &\stackrel{\text{Fact 1}}{\geq} \sum_{k \in C_i} p_k \cdot \left(\frac{1}{4}\right)^{\sum_{k \in C_i} p_k} \geq \frac{1}{2^\beta} \cdot \left(\frac{1}{4}\right)^{\frac{1}{2^\beta}}. \end{aligned}$$

The last inequality holds because the previous function is increasing in  $[\frac{1}{2^\beta}, \frac{3}{2^\beta}]$ .

The probability  $P_c$  that exactly one node in  $C_i$  and no other node in  $D_i$  sends is therefore given by

$$P_c = P_0 \cdot P_{suc} \geq \left[\left(\frac{1}{4}\right)^{\frac{3}{2^\beta}}\right]^{18} \cdot \frac{1}{2^\beta} \left(\frac{1}{4}\right)^{\frac{1}{2^\beta}} \stackrel{\beta=6}{=} \frac{2^{9/32}}{256}.$$

$P_c$  is a lower bound for the probability that  $C_i$  clears itself in a time-slot  $t \in \mathcal{I}'$ . The reason for choosing  $\beta = 6$  is that this value maximizes  $P_c$ . The probability  $\overline{P_{term}}$  that circle  $C_i$  does not clear itself during the entire interval is  $\overline{P_{term}} \leq (1 - 2^{9/32}/256)^\delta \leq n^{-2.3} \in o(n^{-2})$ . We have thus shown that within  $\mathcal{I}'$ , the sum of sending probabilities in  $C_i$  either falls back below  $\frac{1}{2^\beta}$  or  $C_i$  clears itself.

For the induction step, we consider an arbitrary  $t_i^j$ . Due to the induction hypothesis, we can assume that all previous such time-slots have already been dealt with. In other words, all previously considered time-slots  $t_{i'}^{j'}$  have either lead to a clearance of circle  $C_{i'}$  or the sum of probabilities in  $C_{i'}$  has decreased below the threshold  $\frac{1}{2^\beta}$ . Immediately after a clearance, the sum of sending probabilities in a circle  $C_i$  is at most  $\frac{1}{2^\beta}$ , which is the sending probability in the last round of the algorithm. This is true because only one node in the circle remains undecided. All others will stop sending on channel  $\Gamma_1$ . By Lemma 1, the sum of sending probabilities in all neighboring circles (both the cleared and the not cleared ones) is bounded by  $\frac{3}{2^\beta}$  in  $\mathcal{I}_i^j$  (otherwise, this circle would have been considered before  $t_i^j$ ). Therefore, we know that the bounds (1) and (2) hold with high probability and the computation for the induction step is the same as for the base case  $t'$ .

Because there are  $n$  nodes to be decided and at most  $n$  circles  $C_i$ , the number of induction steps  $t_i^j$  is at most  $n$ . Hence, the probability that the claim holds for all steps is at least  $(1 - o(\frac{1}{n^2}))^n \geq 1 - o(\frac{1}{n})$ .

Using Lemma 2, we can now compute the expected number of dominators in each circle  $C_i$ . In the analysis, we will separately compute the number of dominators *before* and *after* the termination (i.e., the first clearance) of  $C_i$ .

**Lemma 3.** *Let  $C$  be the number of collisions (more than one node is sending in one time-slot on  $\Gamma_1$ ) in a circle  $C_i$ . The expected number of collisions in  $C_i$  before its termination is  $E[C] < 6$  and  $C < 7 \log n$  with probability at least  $1 - o(n^{-2})$ .*

*Proof.* Only channel  $\Gamma_1$  is considered in this proof. We assume that  $C_i$  is not yet terminated and we define the following events

- $A$  : Exactly one node in  $D_i$  is sending
- $X$  : More than one node in  $C_i$  is sending
- $Y$  : At least one node in  $C_i$  is sending
- $Z$  : Some node in  $D_i \setminus C_i$  is sending

For the proof, we consider only rounds in which at least one node in  $C_i$  sends. (No new dominators emerge in  $C_i$  if no node sends). We want to bound the conditional probability  $P[A | Y]$  that exactly one node  $v \in C_i$  in  $D_i$  is sending. Using  $P[Y | X] = 1$  and the fact that  $Y$  and  $Z$  are independent, we get

$$\begin{aligned} P[A | Y] &= P[\overline{X} | Y] \cdot P[\overline{Z} | Y] = (1 - P[X | Y]) (1 - P[Z]) \\ &= \left(1 - \frac{P[X]P[Y | X]}{P[Y]}\right) (1 - P[Z]) = \left(1 - \frac{P[X]}{P[Y]}\right) (1 - P[Z]). \quad (4) \end{aligned}$$

We can now compute bounds for the probabilities  $P[X]$ ,  $P[Y]$ , and  $P[Z]$ :

$$\begin{aligned} P[X] &= 1 - \prod_{k \in C_i} (1 - p_k) - \sum_{k \in C_i} \left( p_k \prod_{\substack{l \in C_i \\ l \neq k}} (1 - p_l) \right) \\ &\leq 1 - \left( \frac{1}{4} \right)^{\sum_{k \in C_i} p_k} - \sum_{k \in C_i} p_k \cdot \left( \frac{1}{4} \right)^{\sum_{k \in C_i} p_k} \\ &= 1 - \left( 1 + \sum_{k \in C_i} p_k \right) \left( \frac{1}{4} \right)^{\sum_{k \in C_i} p_k} \end{aligned}$$

The first inequality for  $P[X]$  follows from Fact 1 and inequality (4). Using Fact 1, we can bound  $P[Y]$  as  $P[Y] = 1 - \prod_{k \in C_i} (1 - p_k) \geq 1 - (1/e)^{\sum_{k \in C_i} p_k}$ . In the proof for Lemma 2, we have already computed a bound for  $P_0$ , the probability that no node in  $D_i \setminus C_i$  sends. Using this result, we can write  $P[Z]$  as

$$P[Z] = 1 - \prod_{C_j \in D_i \setminus C_i} \prod_{k \in C_j} (1 - p_k) \stackrel{\text{Eq. (3)}}{\leq} 1 - \left[ \left( \frac{1}{4} \right)^{\frac{3}{2^\beta}} \right]^{18}.$$

Plugging all inequalities into equation (4), we obtain the desired function for  $P[A | Y]$ . It can be shown that the term  $P[X]/P[Y]$  is maximized for  $\sum_{k \in C_i} p_k = \frac{3}{2^\beta}$  and thus,  $P[A | Y] = (1 - P[X]/P[Y]) \cdot (1 - P[Z]) \geq 0.18$ .

This shows that whenever a node in  $C_i$  sends,  $C_i$  terminates with constant probability at least  $P[A | Y]$ . This allows us to compute the expected number of collisions in  $C_i$  before the termination of  $C_i$  as a geometric distribution,  $E[C] = P[A | Y]^{-1} \leq 6$ . The high probability result can be derived as  $P[C \geq 7 \log n] = (1 - P[A | Y])^{7 \log n} \in O(n^{-2})$ .

So far, we have shown that the number of collisions before the clearance of  $C_i$  is constant in expectation. The next lemma shows that the number of *new dominators per collision* is also constant. In a collision, each of the sending nodes may already be dominator. Hence, if we assume that every sending node in a collision is a new dominator, we obtain an upper bound for the true number of new dominators.

**Lemma 4.** *Let  $D$  be the number of nodes in  $C_i$  sending in a time-slot and let  $\Phi$  denote the event of a collision. Given a collision, the expected number of sending nodes is  $E[D | \Phi] \in O(1)$ . Furthermore,  $P[D < \log n | \Phi] \geq 1 - o(\frac{1}{n^2})$ .*

*Proof.* Let  $m$ ,  $m \leq n$ , be the number of nodes in  $C_i$  and  $N = \{1 \dots m\}$ .  $D$  is a random variable denoting the number of sending nodes in  $C_i$  in a given time-slot. We define  $A_k := P[D = k]$  as the probability that exactly  $k$  nodes send. Let  $\binom{N}{k}$  be the set of all  $k$ -subsets of  $N$  (subsets of  $N$  having exactly  $k$  elements).

Defining  $A'_k$  as  $A'_k := \sum_{Q \in \binom{N}{k}} \prod_{i \in Q} \frac{p_i}{1-p_i}$  we can write  $A_k$  as

$$\begin{aligned} A_k &= \sum_{Q \in \binom{N}{k}} \left( \prod_{i \in Q} p_i \cdot \prod_{i \notin Q} (1-p_i) \right) \\ &= \left( \sum_{Q \in \binom{N}{k}} \prod_{i \in Q} \frac{p_i}{1-p_i} \right) \cdot \prod_{i=1}^m (1-p_i) = A'_k \cdot \prod_{i=1}^m (1-p_i). \end{aligned} \quad (5)$$

**Fact 3.** *The following recursive inequality holds between two subsequent  $A'_k$ :*

$$A'_k \leq \frac{1}{k} \sum_{i=1}^m \frac{p_i}{1-p_i} \cdot A'_{k-1} \quad , \quad A'_0 = 1.$$

*Proof.* The probability  $A_0$  that no node sends is  $\prod_{i=1}^m (1-p_i)$  and therefore  $A'_0 = 1$ , which follows directly from equation (5). For general  $A'_k$ , we have to group the terms  $\prod_{i \in Q} \frac{p_i}{1-p_i}$  in such a way that we can factor out  $A'_{k-1}$ :

$$\begin{aligned} A'_k &= \sum_{Q \in \binom{N}{k}} \prod_{j \in Q} \frac{p_j}{1-p_j} = \frac{1}{k} \sum_{i=1}^m \left( \frac{p_i}{1-p_i} \cdot \sum_{Q \in \binom{N \setminus \{i\}}{k-1}} \prod_{j \in Q} \frac{p_j}{1-p_j} \right) \\ &\leq \frac{1}{k} \sum_{i=1}^m \left( \frac{p_i}{1-p_i} \cdot \sum_{Q \in \binom{N}{k-1}} \prod_{j \in Q} \frac{p_j}{1-p_j} \right) \\ &= \frac{1}{k} \sum_{i=1}^m \frac{p_i}{1-p_i} \cdot \left( \sum_{Q \in \binom{N}{k-1}} \prod_{j \in Q} \frac{p_j}{1-p_j} \right) = \frac{1}{k} \sum_{i=1}^m \frac{p_i}{1-p_i} \cdot A'_{k-1}. \end{aligned}$$

We now continue the proof of Lemma 4. The conditional expected value  $E[D \mid \Phi]$  is  $E[D \mid \Phi] = \sum_{i=0}^m (i \cdot P[D = i \mid \Phi]) = \sum_{i=2}^m B_i$  where  $B_i$  is defined as  $i \cdot P[D = i \mid \Phi]$ . For  $i \geq 2$ , the conditional probability reduces to  $P[D = i \mid \Phi] = P[D = i] / P[\Phi]$ . In the next step, we consider the ratio between two consecutive terms of  $\sum_{i=2}^m B_i$ .

$$\begin{aligned} \frac{B_{k-1}}{B_k} &= \frac{(k-1) \cdot P[D = k-1 \mid \Phi]}{k \cdot P[D = k \mid \Phi]} = \frac{(k-1) \cdot P[D = k-1]}{k \cdot P[D = k]} \\ &= \frac{(k-1) \cdot A_{k-1}}{k \cdot A_k} = \frac{(k-1) \cdot A'_{k-1}}{k \cdot A'_k}. \end{aligned}$$

It follows from Fact 3, that each term  $B_k$  can be upper bounded by

$$\begin{aligned} B_k &= \frac{k A'_k}{(k-1) A'_{k-1}} \cdot B_{k-1} \stackrel{\text{Fact 3}}{\leq} \frac{k \left( \frac{1}{k} \sum_{i=1}^m \frac{p_i}{1-p_i} \cdot A'_{k-1} \right)}{(k-1) A'_{k-1}} \cdot B_{k-1} \\ &= \frac{1}{k-1} \sum_{i=1}^m \frac{p_i}{1-p_i} \cdot B_{k-1} \leq \frac{2}{k-1} \sum_{i=1}^m p_i \cdot B_{k-1}. \end{aligned}$$

The last inequality follows from  $\forall i : p_i < 1/2$  and  $p_i \leq 1/2 \Rightarrow \frac{p_i}{1-p_i} \leq 2p_i$ .

From the definition of  $B_k$ , it naturally follows that  $B_2 \leq 2$ . Furthermore, we can bound the sum of sending probabilities  $\sum_{i=1}^m p_i$  using Lemma 2 to be less than  $\frac{3}{2^\beta}$ . We can thus sum up over all  $B_i$  recursively in order to obtain  $E[D \mid \Phi]$ :

$$E[D \mid \Phi] = \sum_{i=2}^m B_i \leq 2 + \sum_{i=3}^m \left[ \frac{2}{(i-1)!} \left( \frac{6}{2^\beta} \right)^{i-2} \right] \leq 2.11.$$

For the high probability result, we solve the recursion of Fact 3 and obtain  $A'_k \leq \frac{1}{k!} \left( \sum_{i=1}^m \frac{p_i}{1-p_i} \right)^k$ . The probability  $P_+ := P[D \geq \log n \mid \Phi]$  is

$$\begin{aligned} P_+ &= \sum_{k=\lceil \log m \rceil}^m A_k \leq \sum_{k=\lceil \log m \rceil}^m A'_k \leq \sum_{k=\lceil \log m \rceil}^m \left[ \frac{1}{k!} \cdot \left( \sum_{i=1}^m \frac{p_i}{1-p_i} \right)^k \right] \\ &\leq \sum_{k=\lceil \log m \rceil}^m \left[ \frac{1}{k!} \cdot \left( 2 \cdot \sum_{i=1}^m p_i \right)^k \right] \stackrel{\text{Lm. 2}}{\leq} (m - \lceil \log m \rceil) \cdot \frac{(2 \cdot \sum_{i=1}^m p_i)^{\lceil \log m \rceil}}{\lceil \log m \rceil!} \\ &\leq m \cdot \left( 2 \cdot \sum_{i=1}^m p_i \right)^{\lceil \log m \rceil} \leq m \cdot \left( \frac{6}{2^\beta} \right)^{\lceil \log m \rceil} \in O\left( \frac{1}{m^2} \right). \end{aligned}$$

The last key lemma shows that the expected number of new dominators *after* the termination of circle  $C_i$  is also constant.

**Lemma 5.** *Let  $A$  be the number of new dominators after the termination of  $C_i$ . Then,  $A \in O(1)$  with high probability.*

*Proof.* We define  $B$  and  $B_i$  as the set of dominators in  $D_i$  and  $C_i$ , respectively. Immediately after the termination of  $C_i$ , only one node in  $C_i$  remains sending on channel  $\Gamma_1$ , because all others will be decided. Because  $C$  and  $D \mid \Phi$  are independent variables, it follows from Lemmas 3 and 4 that  $|B_i| \leq \tau' \log^2 n$  for a small constant  $\tau'$ . Potentially, all  $C_j \in D_i$  are already terminated and therefore,  $1 \leq |B| \leq 19 \cdot \tau \log^2 n$  for  $\tau := 19 \cdot \tau'$ . We distinguish the two cases  $1 \leq |B| \leq \tau \log n$  and  $\tau \log n < |B| \leq \tau \log^2 n$  and consider channels  $\Gamma_2$  and  $\Gamma_3$  in the first and second case, respectively. We show that in either case, a new node will receive a message on one of the two channels with high probability during the algorithm's waiting period.

First, consider case one, i.e.  $1 \leq |B| \leq \tau \log n$ . The probability  $P_0$  that one dominator is sending alone on channel  $\Gamma_2$  is  $P_0 = |B| \cdot q \cdot (1-q)^{|B|-1}$ . This is a concave function in  $|B|$ . For  $|B| = 1$ , we get  $P_0 = q = (2^\beta \cdot \lceil \log n \rceil)^{-1}$  and for  $|B| = \tau \log n$ ,  $n \geq 2$ , we have

$$\begin{aligned} P_0 &= \frac{\tau \log n}{2^\beta \lceil \log n \rceil} \cdot \left( 1 - \frac{1}{2^\beta \lceil \log n \rceil} \right)^{\tau \log n - 1} \geq \frac{\tau}{2^\beta} \cdot \left( 1 - \frac{\tau/2^\beta}{\tau \log n} \right)^{\tau \log n} \\ &\stackrel{\text{Fact 2}}{\geq} \frac{\tau}{2^\beta} e^{-\frac{\tau}{2^\beta}} \left( 1 - \frac{(\tau/2^\beta)^2}{\tau \log n} \right) \stackrel{(n \geq 2)}{\geq} \frac{\tau}{2^\beta} e^{-\frac{\tau}{2^\beta}} \left( 1 - \frac{\tau}{2 \cdot 2^\beta} \right) \in O(1). \end{aligned}$$

A newly awakened node in a terminated circle  $C_i$  will not send during the first  $\delta \cdot \lceil \log \hat{n} \rceil$  rounds. If during this period, the node receives a message from an existing dominator, it will become decided and hence, will not become dominator. The probability that such an already covered node does *not* receive any messages from an existing dominator is bounded by  $P_{no} \leq (1 - (2^\beta \cdot \lceil \log n \rceil)^{-1})^{\delta \cdot \lceil \log n \rceil} \leq e^{-\delta/2^\beta} \in O(n^{-7})$ . Hence, with high probability, the number of new dominators is bounded by a constant in this case. The analysis for the second case follows along the same lines (using  $\Gamma_3$  instead of  $\Gamma_2$ ) and is omitted.

Finally, we formulate and prove the main theorem.

**Theorem 2.** *The algorithm computes a correct dominating set in time  $O(\log^2 \hat{n})$  and achieves an approximation ratio of  $O(1)$  in expectation.*

*Proof.* Correctness and running time follow from Theorem 1. For the approximation ratio, consider a circle  $C_i$ . The expected number of dominators in  $C_i$  before the termination of  $C_i$  is  $E[D] = E[C] \cdot E[D | \Phi] \in O(1)$  by Lemmas 3 and 4. By Lemma 5, the number of dominators emerging after the termination of  $C_i$  is also constant. The Theorem follows from the fact that the optimal solution must choose at least one dominator in  $D_i$ .

## References

1. D. J. Baker and A. Ephremides. The Architectural Organization of a Mobile Radio Network via a Distributed Algorithm. *IEEE Trans. Communications*, COM-29(11):1694–1701, 1981.
2. R. Bar-Yehuda, O. Goldreich, and A. Itai. On the Time-Complexity of broadcast in radio networks: an exponential gap between determinism randomization. In *Proc. 6<sup>th</sup> Symposium on Principles of Distributed Computing (PODC)*, pages 98–108. ACM Press, 1987.
3. U. Feige. A Threshold of  $\ln n$  for Approximating Set Cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.
4. J. Gao, L. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Discrete Mobile Centers. In *Proc. 17<sup>th</sup> Symposium on Computational Geometry (SCG)*, pages 188–196. ACM Press, 2001.
5. L. Gasieniec, A. Pelc, and D. Peleg. The wakeup problem in synchronous broadcast systems (extended abstract). In *Proc. 19<sup>th</sup> Symposium on Principles of Distributed Computing (PODC)*, pages 113–121. ACM Press, 2000.
6. M. Gerla and J. Tsai. Multicluster, mobile, multimedia radio network. *ACM/Baltzer Journal of Wireless Networks*, 1(3):255–265, 1995.
7. L. Jia, R. Rajaraman, and R. Suel. An Efficient Distributed Algorithm for Constructing Small Dominating Sets. In *Proc. of the 20<sup>th</sup> ACM Symposium on Principles of Distributed Computing (PODC)*, pages 33–42, 2001.
8. T. Jurdzinski and G. Stachowiak. Probabilistic Algorithms for the Wakeup Problem in Single-Hop Radio Networks. In *Proc. 13<sup>th</sup> Int. Symposium on Algorithms and Computation (ISAAC)*, volume 2518 of *Lecture Notes in Computer Science*, pages 535–549, 2002.
9. F. Kuhn and R. Wattenhofer. Constant-Time Distributed Dominating Set Approximation. In *Proc. 22<sup>nd</sup> Symp. on Principles of Distributed Computing (PODC)*, pages 25–32, 2003.
10. S. Kutten and D. Peleg. Fast Distributed Construction of Small k-Dominating Sets and Applications. *Journal of Algorithms*, 28:40–66, 1998.
11. L. G. Roberts. Aloha Packet System with and without Slots and Capture. *ACM SIGCOMM, Computer Communication Review*, 5(2):28–42, 1975.
12. P. Wan, K. Alzoubi, and O. Frieder. Distributed construction of connected dominating set in wireless ad hoc networks. In *Proceedings of INFOCOM*, 2002.



# Seeking a Vertex of the Planar Matching Polytope in NC

Raghav Kulkarni<sup>1\*</sup> and Meena Mahajan<sup>2</sup>

<sup>1</sup> Chennai Mathematical Institute, 92, G.N.Chetty Road, T. Nagar, Chennai 600 017, India. [raghav@cmi.ac.in](mailto:raghav@cmi.ac.in)

<sup>2</sup> The Institute of Mathematical Sciences, Chennai 600 113, India. [meena@imsc.res.in](mailto:meena@imsc.res.in)

**Abstract.** For planar graphs, counting the number of perfect matchings (and hence determining whether there exists a perfect matching) can be done in NC [4,10]. For planar bipartite graphs, finding a perfect matching when one exists can also be done in NC [8,7]. However in general planar graphs (when the bipartite condition is removed), no NC algorithm for constructing a perfect matching is known.

We address a relaxation of this problem. We consider the fractional matching polytope  $\mathcal{P}(G)$  of a planar graph  $G$ . Each vertex of this polytope is either a perfect matching, or a half-integral solution: an assignment of weights from the set  $\{0, 1/2, 1\}$  to each edge of  $G$  so that the weights of edges incident on each vertex of  $G$  add up to 1 [6]. We show that a vertex of this polytope can be found in NC, provided  $G$  has at least one perfect matching to begin with. If, furthermore, the graph is bipartite, then all vertices are integral, and thus our procedure actually finds a perfect matching without explicitly exploiting the bipartiteness of  $G$ .

## 1 Introduction

The perfect matching problem is of fundamental interest to combinatorists, algorithmists and complexity-theorists for a variety of reasons. In particular, the problems of deciding if a graph has a perfect matching, and finding such a matching if one exists, have received considerable attention in the field of parallel algorithms. Both these problems are in randomized NC [5,2,9] but are not known to be in deterministic NC. (NC is the class of problems with parallel algorithms running in polylogarithmic time using polynomially many processors.) For special classes of graphs, however, there are deterministic NC algorithms.

In this work, we focus on planar graphs. These graphs are special for the following reason: for planar graphs, we can count the number of perfect matchings in NC ([4,10], see also [3]), but we do not yet know how to find one, if one exists. This counters our intuition that decision and search versions of problems are easier than their counting versions. In fact, for the perfect matching problem

---

\* Part of this work was done when this author was visiting the Institute of Mathematical Sciences, Chennai on a summer student programme.

itself, while decision and search are both in P, counting is #P-hard and hence is believed to be much harder. But for planar graphs the situation is curiously inverted.

We consider a relaxation of the search version and show that it admits a deterministic NC algorithm. The problem we consider is the following: For a graph  $G = (V, E)$  with  $|V| = n$  vertices and  $|E| = m$  edges, consider the  $m$ -dimensional space  $\mathcal{Q}^m$ . A point  $\langle x_1, x_2, \dots, x_m \rangle$  in this space can be viewed as an assignment of weight  $x_i$  to the  $i$ th edge of  $G$ . A perfect matching in  $G$  is such an assignment (matched edges have weight 1, other edges have weight 0) and hence is a point in this space.

Consider the polytope  $\mathcal{P}(G)$  defined by the following equations.

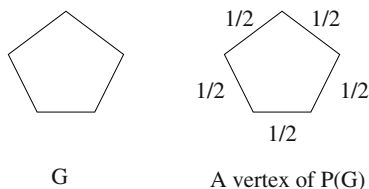
$$x_e \geq 0 \quad \forall e \in E \tag{1}$$

$$\sum_{e \text{ incident on } v} x_e = 1 \quad \forall v \in V \tag{2}$$

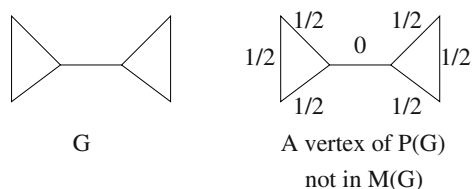
Clearly, every perfect matching of  $G$  (i.e. the corresponding point in  $\mathcal{Q}^m$ ) lies in  $\mathcal{P}(G)$ . Standard matching theory ( see for instance [6]) tells us that every perfect matching of  $G$  is a vertex of  $\mathcal{P}(G)$ . In fact, if we denote by  $\mathcal{M}(G)$  the convex hull of all perfect matchings, then  $\mathcal{M}(G)$  is a facet of  $\mathcal{P}(G)$ . In the case of bipartite graphs, the perfect matchings are in fact the only vertices of  $\mathcal{P}(G)$ ;  $\mathcal{P}(G) = \mathcal{M}(G)$ . Thus for a bipartite graph it suffices to find a vertex of  $\mathcal{P}(G)$  to get a perfect matching. Furthermore, for general graphs, all vertices of  $\mathcal{P}(G)$  are always half-integral (in the set  $\{0, 1/2, 1\}^m$ ). For any vertex  $w$  of  $\mathcal{P}(G)$ , if we pick those edges of  $G$  having non-zero weight in  $w$ , we get a subgraph which is a disjoint union of a partial matching and some odd cycles.

For instance, Figure 1 shows a graph where  $\mathcal{M}(G)$  is empty, while  $\mathcal{P}(G)$  has one point shown by the weighted graph alongside. Figure 2 shows a graph where  $\mathcal{M}(G)$  is non-empty; furthermore, the weighted graph in Figure 2(b) is a vertex of  $\mathcal{P}(G)$  not in  $\mathcal{M}(G)$ .

We show in this paper that finding a vertex of  $\mathcal{P}(G)$  is in NC when  $G$  is a planar graph with at least one perfect matching. If, furthermore, the graph is bipartite, then all vertices are integral, and thus our procedure actually finds a perfect matching, without explicitly exploiting the bipartiteness of  $G$ .



**Fig. 1.** An Example Graph with empty  $\mathcal{M}(G)$ , and a point in  $\mathcal{P}(G)$



**Fig. 2.** An Example Graph with non-empty  $\mathcal{M}(G)$ , and a point in  $\mathcal{P}(G) - \mathcal{M}(G)$

Our approach is as follows. In [7], an NC procedure is described to find a point  $p$  inside  $\mathcal{P}(G)$ , using the fact that counting the number of perfect matchings in a planar graph is in NC. Then, exploiting the planarity and bipartiteness of the graph, an NC procedure is described to find a vertex of  $\mathcal{P}(G)$  by constructing a path, starting from  $p$ , that never leaves  $\mathcal{P}(G)$  and that makes measurable progress towards a vertex. We extend the same approach for general planar graphs. In fact, we start at the same point  $p$  found in [7]. Then, using only planarity of  $G$ , we construct a path, starting from  $p$  and moving towards a vertex of  $\mathcal{P}(G)$ . Our main contribution, thus, is circumventing the bipartite restriction. We prove that our algorithm indeed reaches a vertex of  $\mathcal{P}(G)$  and can be implemented in NC.

This paper is organised as follows. In Section 2 we briefly describe the Mahajan-Varadarajan algorithm [7]. In Section 3 we describe our generalisation of their algorithm, and in the subsequent two sections we prove the correctness and the NC implementation of our method.

## 2 Reviewing the Mahajan-Varadarajan Algorithm

The algorithm of Mahajan & Varadarajan [7] is quite elegant and straightforward and is summarised below.

For planar graphs, the number of perfect matchings can be found in NC [4, 10]. Using this procedure as a subroutine, and viewing each perfect matching as a point in  $\mathcal{Q}^m$ , obtain the point  $p$  which is the average of all perfect matchings; clearly, this point is inside  $\mathcal{M}(G)$ . Now construct a path from  $p$  to some vertex of  $\mathcal{M}(G)$ , always staying inside  $\mathcal{M}(G)$ .

*The Algorithm.*

- Find a point  $p \in \mathcal{M}(G)$ :  $\forall e \in E$ , assign  $x_e = (\#G - \#G_e)/\#G$ , where  $\#G$  denotes the number of perfect matchings in  $G$  and  $G_e$  denotes the graph obtained by deleting edge  $e$  from  $G$ . Delete edges of 0 weight.
- While the graph is not acyclic,
  - Get  $cf$  edge-disjoint cycles (where  $f$  is the number of faces in a planar embedding of  $G$ ) using the fact that the number of edges is less than three times the number of vertices. (Here  $c$  is a constant  $c = 1/24$ . We consider that subgraph of the dual containing faces with fewer than 12 bounding edges. A maximal independent set in this graph is sufficiently large, and gives the desired set of edge-disjoint cycles.) Since the graph is bipartite, all the cycles are of even length.
  - Destroy each of these cycles by removing the smallest weight edge in it. To stay inside the polytope, manipulate the weights of the remaining edges as follows: in a cycle  $C$ , if  $e$  is the smallest weight edge with weight  $x_e$ , then add  $x_e$  to the weight of edges at odd distances from  $e$  and subtract  $x_e$  from the weights of edges at even distances from  $e$ . Thus the edge  $e$  itself gets weight 0. Delete all 0-weight edges.

- Finally we get an acyclic graph and we are still inside the polytope. It's easy to check that any acyclic graph inside  $\mathcal{M}(G)$  must be a perfect matching (integral).

Since we are destroying  $cf$  faces each time, within  $\log f$  iterations of the while loop we will end up with an acyclic graph. Hence, for bipartite planar graphs finding a perfect matching is in NC.

### 3 Finding a Half-Integral Solution in a Planar Graph in NC

The following result is a partial generalization of the previous result.

**Theorem 1.** *For planar graphs, a vertex of the fractional matching polytope  $\mathcal{P}(G)$  (i.e. a half-integral solution to the equations defining  $\mathcal{P}(G)$ , with no even cycles) can be found in NC, provided that the perfect matching polytope  $\mathcal{M}(G)$  is non-empty.*

Our starting point is the same point  $p$  computed in the previous section; namely, the arithmetic mean of all perfect matchings of  $G$ . Starting from  $p$ , we attempt to move towards a vertex. The basic strategy is to find a large set  $S$  of edge-disjoint faces. Each such face contains a simple cycle, which we try to destroy. Difficulties arise if the edges bounding the faces in  $S$  do not contain even length simple cycles, since the method of the previous section works only for even cycles. We describe mechanisms to be used successively in such cases.

We first describe some basic building blocks, and then describe how to put them together.

*Building Block 1:* Simplify, or *Standardise*, the graph  $G$ .

Let  $G$  be the current graph, let  $x : E \rightarrow \mathcal{Q}$  be the current assignment of weights to edges, and let  $y$  be the partial assignment finalised so far. The final assignment is  $y : E \rightarrow \{0, 1/2, 1\}$ .

**Step 1.1.** For each  $e = (u, v) \in E(G)$ , if  $x_e = 0$ , then set  $y_e = 0$  and delete  $e$  from  $G$ .

**Step 1.2.** For each  $e = (u, v) \in E(G)$ , if  $x_e = 1$ , then set  $y_e = 1$  and delete  $u$  and  $v$  from  $G$ .

(This step ensures that all vertices of  $G$  have degree at least 2.)

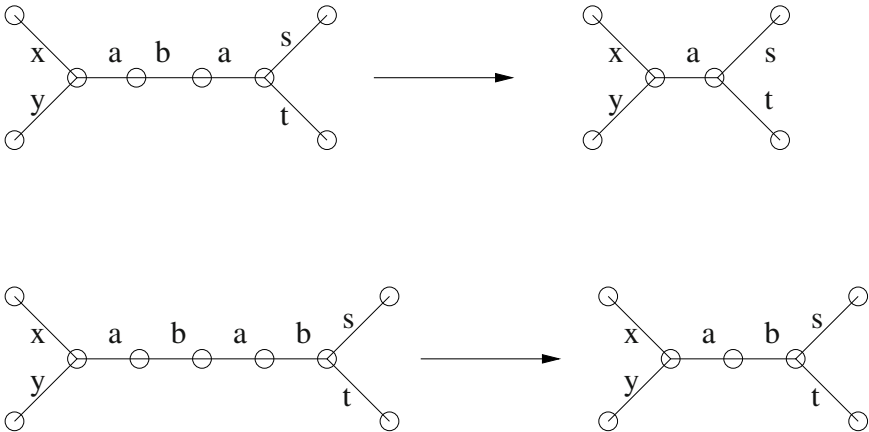
**Step 1.3.** Obtain connected components of  $G$ .

If a component is an odd cycle  $C$ , then every edge on  $C$  must have weight  $1/2$ . For each  $e \in C$ , set  $y_e = 1/2$ . Delete all the edges and vertices of  $C$  from  $G$ .

If a component is an even cycle  $C$ , then for some  $0 < a < 1$ , the edges on  $C$  alternately have weights  $a$  and  $1 - a$ . For each  $e \in C$ , if  $x_e = a$  then set  $y_e = 1$  and if  $x_e = 1 - a$  then set  $y_e = 0$ . Delete all the edges and vertices of  $C$  from  $G$ .

**Step 1.4.** Let  $E'$  be the set of edges touching a vertex of degree 2 in  $G$ . Consider the subgraph of  $G$  induced by  $E'$ ; this is a disjoint collection of paths. Collapse each such even path to a path of length 2 and each such odd path to a path of length 1, reassigning weights as shown in Figure 3. Again, we stay within the polytope of the new graph, and from any assignment here, a point in  $\mathcal{P}(G)$  can be recovered in a straightforward way.

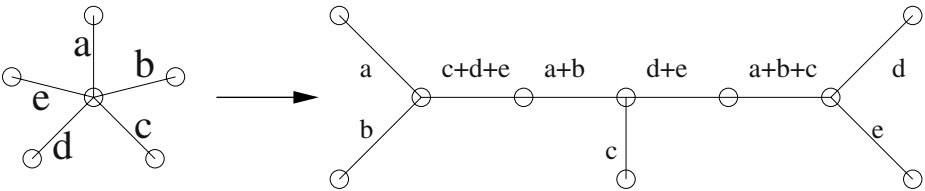
This step ensures that no degree 2 vertex has a degree 2 neighbour.



**Fig. 3.** Transformation assuring that no two consecutive vertices are of degree 2

**Step 1.5.** For each  $v \in V(G)$ , if  $v$  has degree more than 3, then introduce some new vertices and edges, rearrange the edges touching  $v$ , and assign weights as shown in Figure 4. This assignment in the new graph is in the corresponding polytope of the new graph, and from any assignment here, a point in  $\mathcal{P}(G)$  can be recovered in a straightforward way. (This gadget construction was in fact first used in [1].)

This step ensures that all vertices have degree 2 or 3.



**Fig. 4.** Transformation to remove vertices of degree greater than 3

Note that Steps 1.4 and 1.5 above change the underlying graph. To recover the point in  $\mathcal{P}(G)$  from a point in the new graph's polytope, we can initially allocate one processor per edge. This processor will keep track of which edge in the modified graph dictates the assignment to this edge. Whenever any transformation is done on the graph, these processors update their respective data, so that recovery at the end is possible.

We will call a graph on which the transformations of building block 1 have been done a *standardised graph*.

*Building Block 2:* Process an even cycle. This is as in [7], and is described in Section 2.

*Building Block 3:* Process an odd cycle connected to itself by a path. Let  $C$  be such an odd cycle, with path  $P$  connecting  $C$  to itself. We first consider the case when  $P$  is a single edge, i.e. a chord. The chord  $(u, v)$  cuts the cycle into paths  $P_1, P_2$ . Let  $C_i$  denote the cycle formed by  $P_i$  along with the chord  $(u, v)$ . Exactly one of  $C_1, C_2$  is even; process it as in Building Block 2.

If instead of a chord, there is some path  $P_{u,v}$  connecting  $u$  and  $v$  on  $C$ , the same reasoning holds and so this step can still be performed.

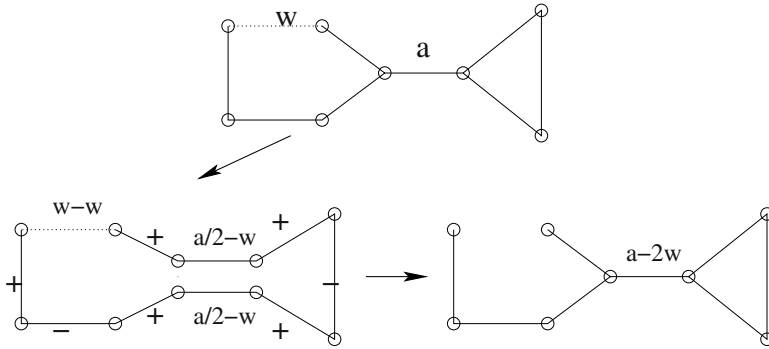
*Building Block 4:* Process a pair of edge-disjoint odd cycles connected by a path.

Let  $C_1$  and  $C_2$  be the odd cycles and  $P$  the path connecting them. Note that if  $G$  is standardised, then  $P$  cannot be of length 0. Let  $P$  connect to  $C_1$  at  $u$  and to  $C_2$  at  $v$ . Then the traversal of  $C_1$  beginning at  $u$ , followed by path  $P$  going from  $u$  to  $v$ , then the traversal of  $C_2$  beginning at  $v$ , followed by the path  $P$  going from  $v$  to  $u$ , is a closed walk of even length. We make two copies of  $P$ , one for each direction of traversal. For edge  $e$  on  $P$ , assign weight  $x_e/2$  to each copy. Now treating the two copies as separate, we have an even cycle which can be processed according to building Block 2. For each edge  $e \in P$ , its two copies are at even distance from each other, so either both increase or both decrease in weight. It can be seen that after this adjustment, the weights of the copies put together is still between 0 and 1.

This step is illustrated in Figure 5. The edge on the path has weight  $a$  is split into two copies with weight  $a/2$  each. The dotted edge is the minimum weight edge; thus  $w \leq a/2$ .

*The Algorithm.* The idea is to repeatedly identify large sets of edge-disjoint faces, and then manipulate them, in the process destroying them. The faces are identified as in [7], and a simple cycle is extracted from each face. Even cycles are processed using Building Block 2. By the building blocks 3 and 4, odd cycles can also be processed provided we identify paths connecting the cycles to themselves or other cycles. However, to achieve polylogarithmic time, we need to process several odd cycles simultaneously, and this requires that the odd cycles and the connecting paths be edge-disjoint.

We use the following definition: A path  $P$  is said to be a *3-bounded* path if the number of internal vertices of  $P$  with degree 3 is at most 1. Note that in a standardised graph, a 3-bounded path can have at most 3 internal vertices.



**Fig. 5.** Manipulating a closed walk of even length

The algorithm can be described as follows:

1. Find a point  $p \in \mathcal{M}(G)$  and initialise  $x_e$  accordingly.
2. Standardise  $G$  (Building block 1; this builds a partial solution in  $y$ ).
3. While  $G$  is not empty, repeat the following steps, working in parallel on each connected component of  $G$ :
  - a) Find a collection  $S$  of edge-disjoint faces in  $G$ , including at least  $1/24$  of the faces from each component. Extract a simple cycle from the edges bounding each face of  $S$ , to obtain a collection of simple cycles  $T$ .
  - b) Process all even cycles (Building block 2). Remove these cycles from  $T$ . Re-standardise.
  - c) Define each surviving cycle in  $T$  to be a cluster. (At later stages, a cluster will be a set of vertices and the subgraph induced by this subset.) While  $T$  is non-empty, repeat the following steps:
    - i. Construct an auxiliary graph  $H$  with clusters as vertices.  $H$  has an edge between clusters  $D_1$  and  $D_2$  if there is a 3-bounded path between some vertex of  $D_1$  and some vertex of  $D_2$  in  $G$ .
    - ii. Process all clusters having a self-loop in  $H$ . (Building Block 3). Remove these clusters from  $T$ . Re-standardise.
    - iii. Recompute  $H$ . In  $H$ , find a maximal matching. Each matched edge pairs two clusters, between which there is a 3-bounded path in  $G$ . In parallel, process all these pairs along with the connecting path using Building Block 4. Remove the processed clusters from  $T$  and re-standardise.
    - iv. “Grow” each cluster: if  $D$  is the set of vertices in a cluster, then first add to  $D$  all degree-2 neighbours of  $D$ , then add to  $D$  all degree-3 neighbours of  $D$ . That is,
$$D = D \cup \{v \mid d(v) = 2 \wedge \exists u \in D, (u, v) \in E\},$$

$$D = D \cup \{v \mid d(v) = 3 \wedge \exists u \in D, (u, v) \in E\}.$$
4. Return the edge weights stored in  $y$ .

## 4 Correctness

The correctness of the algorithm follows from the following series of lemmas:

**Lemma 1.** *The clusters and 3-bounded paths processed in Step 3(c) are vertex-disjoint.*

*Proof.* Each iteration of the while loop in Step 3(c) operates on different parts of  $G$ . We show that these parts are edge-disjoint, and in fact even vertex-disjoint. Clearly, this is true when we enter Step 3(c); the cycles are edge-disjoint by our choice in Step 3(a), and since  $G$  has maximum degree 3, no vertex can be on two edge-disjoint cycles. Changes happen only in steps 3(c)(ii) and 3(c)(iii); we analyze these separately.

Consider Step 3(c)(ii). If two clusters  $D_1$  and  $D_2$  have self-loops, then there are 3-bounded paths  $\rho_i$  from  $D_i$  to  $D_i$ ,  $i = 1, 2$ . If these paths share a vertex  $v$ , it can only be an internal vertex of  $\rho_1$  and  $\rho_2$ , since the clusters were vertex-disjoint before this step. In particular,  $v$  cannot be a degree-2 vertex. But since  $G$  is standardized,  $\deg(v)$  is then 3, which does not allow it to participate in two such paths. So  $\rho_1$  and  $\rho_2$  must be vertex-disjoint. Thus processing them in parallel via building block 4 is valid. Processing clusters with self-loops merely removes them from  $T$ ; thus the clusters surviving after Step 3(c)(ii) continue to be vertex-disjoint.

Now consider Step 3(c)(iii). Suppose cluster  $D_1$  is matched to  $D_2$  via 3-bounded path  $\rho$ ,  $D_3$  to  $D_4$  via 3-bounded path  $\eta$ . Note that  $D_i \neq D_j$  for  $i \neq j$ , since we are considering a matching in  $H$ . Thus, by the same argument as above, the paths  $\rho$  and  $\eta$  must be vertex-disjoint. Thus processing them in parallel via building block 4 is valid. Processing matched clusters removes them from  $T$ ; the remaining clusters continue to be vertex-disjoint.

Since Step 3(c)(iii) considers a maximal matching, the clusters surviving are not only vertex-disjoint but also not connected to each other by any 3-bounded path.  $\square$

**Lemma 2.** *Each invocation of the while loop inside Step 3(c) terminates in finite time.*

*Proof.* To establish this statement, we will show that clusters which survive Steps 3(c)(ii) and 3(c)(iii) grow appreciably in size. In particular, they double in each iteration of the while loop. Clearly, clusters cannot double indefinitely while remaining vertex-disjoint, so the statement follows. In fact, our proof establishes that the while loop in Step 3(c) executes  $O(\log n)$  times on each invocation.

Let  $G$  denote the graph at the beginning of Step 3(c). Consider a cluster at this point. Let  $D_0$  be the set of vertices in the cluster. Consider the induced subgraph  $GD_0$  on  $D_0$ . Notice that each such  $GD_0$  contains exactly one cycle, which is an odd cycle extracted in Step 3(a).

We trace the progress of cluster  $D_0$ . Let  $D_i$  denote the cluster (or the associated vertex set; we use this notation interchangeably to mean both) resulting from  $D_0$  after  $i$  iterations of the while loop of Step 3(c). If  $D_0$  does not survive  $i$  iterations, then  $D_i$  is empty.



For any cluster  $D$ , let  $\text{3-size}(D)$  denote the number of vertices in  $D$  whose degree in  $G$  is 3. Let  $D'$  denote the vertices of  $D$  whose degree in  $G$  is 3 but degree in  $D$  is 1.

We establish the following claim:

*Claim.* For a cluster  $D_0$  surviving  $i + 1$  iterations of the while loop of Step 3(c),  $GD_i$  contains exactly one cycle, and furthermore,

$$\begin{aligned} |D'_i| &\geq \lfloor 2^{i-1} \rfloor \\ \text{3-size}(D_i) &\geq 2^i \end{aligned}$$

*Proof of Claim:* As mentioned earlier,  $GD_0$  contains exactly one cycle. Thus  $\text{3-size}|D'_0| = 0$ . In fact, each  $GD_j$ ,  $j \leq i$  contains just this one cycle, because if any other cycle were present in  $GD_j$ , then a self-loop would be found at the  $(j + 1)$ th stage and the cluster would have been processed and deleted from  $T$  in Step 3(c)(ii); it would not grow  $(i + 1)$  times.

It remains to establish the claims on the sizes of  $D_i$  and  $D'_i$ . We establish these claims explicitly for  $i \leq 1$ , and by induction for  $i > 1$ .

Consider  $i = 0$ . Clearly,  $\text{3-size}(D_0) \geq 2^0 = 1$ , and  $\lfloor 2^{-1} \rfloor = 0$ .

Now consider  $i = 1$ . We know that  $D_0$  has gone through two “Grow” phases, and that  $GD_1$  has only one cycle. Notice that each degree 3 vertex in  $D_0$  contributes one vertex *outside*  $D_0$ ; if its third non-cycle neighbour were also on the cycle, then the cycle has a chord detected in Step 3(c)(ii) and  $D_0$  does not grow even once. In fact, since  $D_0$  grows twice, the neighbours are not only outside the cycle but are disjoint from each other. Thus for each vertex contributing to  $\text{3-size}(D_0)$ , one degree-3 vertex is added to  $D_1$  and these vertices are distinct. Thus all these vertices are in  $D'_1$  giving  $|D'_1| = \text{3-size}(D_0) \geq 1$ , and  $\text{3-size}(D_1) = \text{3-size}(D_0) + |D'_1| \geq 2$ .

To complete the induction, assume that the claim holds for  $i - 1$ , where  $i > 1$ . In this case,  $\lfloor 2^{i-2} \rfloor = 2^{i-2}$ . Thus  $\text{3-size}(D_{i-1}) \geq 2^{i-1}$ , and  $|D'_{i-1}| \geq 2^{i-2}$ .

Each  $u \in D'_{i-1}$  has two neighbours,  $u_1$  and  $u_2$ , not in  $D_{i-1}$ . These vertices contributed by each member of  $D'_{i-1}$  must be disjoint, since otherwise  $D_{i-1}$  would have a 3-bounded path to itself and would be processed at the  $i$ th stage; it would not grow the  $i$ th time. Furthermore, if  $u_l$  is of degree 2, let  $u'_l$  denote its degree-3 neighbour other than  $u$ ; otherwise let  $u'_l = u_l$ . By the same reasoning, the vertices  $u'_1, u'_2$  contributed by each  $u \in D'_{i-1}$  must also be disjoint. So  $2|D'_{i-1}|$  vertices are added to  $D_{i-1}$  in obtaining  $D_i$ . All these new vertices must be in  $D'_i$  as well, since otherwise  $D_i$  would have a 3-bounded path to itself and would be processed at the  $(i + 1)$ th stage; it would not grow the  $(i + 1)$ th time. Hence  $|D'_i| = 2|D'_{i-1}| \geq 2 \cdot 2^{i-2} = 2^{i-1}$ .

Every degree-3 vertex of  $D_{i-1}$  continues to be in  $D_i$  and contributes to  $\text{3-size}(D_i)$ . Furthermore, all the vertices of  $D'_i$  are not in  $D_{i-1}$  and also contribute to  $\text{3-size}(D_i)$ . Thus  $\text{3-size}(D_i) = \text{3-size}(D_{i-1}) + |D'_i| \geq 2^{i-1} + 2^{i-1} = 2^i$ .  $\square$

$\square$

**Lemma 3.** *The while loop of Step 3 terminates in finite time.*

*Proof.* Suppose some iteration of the while loop in Step 3 does not delete any edge. This means that Step 3(b) does nothing, so  $S$  has no even cycles, and Step 3(c) deletes nothing, so the clusters keep growing. But by the preceding claim, the clusters can grow at most  $O(\log n)$  times; beyond that, either Step 3(c)(ii) or Step 3(c)(iii) must get executed.

Thus each iteration of the while loop of Step 3 deletes at least one edge from  $G$ , so the while loop terminates in finite time.

**Lemma 4.** *After step 2, and after each iteration of the while loop of Step 3, we have a point inside  $\mathcal{P}(G)$ .*

*Proof.* It is easy to see that all the building blocks described in Section 3 preserve membership in  $\mathcal{P}(G)$ . Hence the point obtained after Step 2 is clearly inside  $\mathcal{P}(G)$ . During Step 3, various edges are deleted by processing even closed walks. By our choice of  $S$ , the even cycles processed simultaneously in Step 3(b) are edge-disjoint. By lemma 1, the even closed walks processed simultaneously in Steps 3(c)(ii) and 3(c)(iii) are edge-disjoint. Now all the processing involves applying one of the building blocks, and these blocks preserve membership in  $\mathcal{P}(G)$  even if applied simultaneously to edge-disjoint even closed walks. The statement follows.  $\square$

**Lemma 5.** *When the algorithm terminates, we have a vertex of  $\mathcal{P}(G)$ .*

*Proof.* When  $G$  is empty, all edges of the original graph have edge weights in the set  $\{0, 1/2, 1\}$ . Consider the graph  $H$  induced by non-zero edge weights  $y_e$ . From the description of Building Block 1, it follows that  $H$  is a disjoint union of a partial matching (with edge weights 1) and odd cycles (with edge weights  $1/2$ ). Such a graph must be a vertex of  $\mathcal{P}(G)$  (see, for instance, [6]).  $\square$

## 5 Analysis

It is clear that each of the basic steps of the algorithm runs in NC. The proof of Lemma 2 establishes that the while loop inside Step 3(c) runs  $O(\log n)$  times. To show that the overall algorithm is in NC, it thus suffices to establish the following:

**Lemma 6.** *The while loop of Step 3 runs  $O(\log n)$  times.*

*Proof.* Let  $F$  be the maximum number of faces per component of  $G$  at the beginning of step 3. We show that  $F$  decreases by a constant fraction after each iteration of the while loop of step 3. Since  $F = O(n)$  for planar graphs, it will follow that the while loop executes at most  $O(\log n)$  times.

At the start of Step 3, connected components of  $G$  are obtained, and they are all handled in parallel. Let us concentrate on any one component. Within each component, unless the component size (and hence number of faces  $f$  in the

embedding of this component) is very small,  $O(1)$ , a set of  $\Omega(f)$  edge-disjoint faces (in fact,  $f/24$  faces) and  $\Omega(f)$  edge-disjoint simple cycles can be found in NC. This is established in Lemma 3 of [7], which basically shows that a maximal independent set amongst the low-degree vertices of the dual is the required set of faces.

So let  $T$  be the set of edge-disjoint faces obtained at the beginning of step 3. If  $|T| \leq f/24$ , then the component is very small, and it can be processed sequentially in  $O(1)$  time. Otherwise, note that after one iteration of the while loop of Step 3,  $T$  is emptied out, so all the clusters in  $T$  get processed. A single processing step handles either one cluster (cluster with self-loop) or two (clusters matched in  $H$ ), so at least  $|T|/2$  processing steps are executed (not necessarily sequentially).

Each processing step deletes at least one edge. Let the number of edges deleted be  $k \geq |T|/2$ . Of these,  $k_1$  are not bridges at the time when they are deleted and  $k_2 = k - k_1$  are bridges when they are deleted. Each deletion of a non-bridge merges two faces in the graph. Thus if  $k_1 \geq |T|/4$ , then at least  $|T|/4 \geq f/96$  faces are deleted; the number of faces decreases by a constant fraction. If  $k_2 > |T|/4$ , consider the effect of these deletions. Each bridge deleted is on a path joining two clusters. Deleting it separates these two clusters into two different components. Thus after  $k_2$  such deletions, we have at least  $k_2$  pairs of separated clusters. Any connected component in the resulting graph has at most one cluster from each pair, and hence *does not have* at least  $k_2$  clusters. Since each cluster contains an odd cycle and hence a face, the number of faces in the new component is at most  $f - k_2 \leq f - |T|/4 \leq f - f/96$ . Hence, either way, the new  $F$  is at most  $95F/96$ , establishing the lemma.  $\square$

## 6 Discussion

We show that if the perfect matching polytope  $\mathcal{M}(G)$  of a planar graph is non-empty, then finding a vertex of  $\mathcal{P}(G)$  is in NC. Unfortunately, we still do not know any way of navigating from a vertex of  $\mathcal{P}(G)$  to a vertex of  $\mathcal{M}(G)$ . Note that both our algorithm and that of [7] navigate “outwards”, in a sense, from the interior of  $\mathcal{P}(G)$  towards an extremal point. To use this to construct a perfect matching in NC, we need a procedure that navigates “along the surface” from a vertex of  $\mathcal{P}(G)$  to a vertex of its facet  $\mathcal{M}(G)$ .

The work of [7] shows that a vertex of  $\mathcal{M}(G)$  can be found not only for bipartite planar graphs but also for bipartite small  $O(\log n)$  genus graphs. The ideas needed to extend the planar bipartite case to the small-genus bipartite case apply here as well; thus we have an NC algorithm for finding a vertex of  $\mathcal{P}(G)$  for  $O(\log n)$  genus graphs.

For perfect matchings in general graphs, search reduces to counting, since search is in P while counting is  $\#P$ -hard even under NC reductions. We believe that this holds for many reasonable subclasses of graphs as well; searching for a perfect matching in a graph reduces, via NC reductions, to counting perfect

matchings in a related graph from the same subclass. Proving this at least for planar graphs would be an important first step.

## References

1. E Dahlhaus and M Karpinski. Perfect matching for regular graphs is  $AC^0$ -hard for the general matching problem. *Journal of Computer and System Sciences*, 44(1):94–102, 1992.
2. R M Karp, E Upfal, and A Wigderson. Constructing a perfect matching is in random NC. *Combinatorica*, 6:35–48, 1986.
3. Marek Karpinski and Wojciech Rytter. *Fast parallel algorithms for graph matching problems*. Oxford University Press, Oxford, 1998. Oxford Lecture Series in Mathematics and its Applications 9.
4. P W Kastelyn. Graph theory and crystal physics. In F Harary, editor, *Graph Theory and Theoretical Physics*, pages 43–110. Academic Press, 1967.
5. L Lovasz. On determinants, matchings and random algorithms. In L Budach, editor, *Proceedings of Conference on Fundamentals of Computing Theory*, pages 565–574. Akademie-Verlag, 1979.
6. L Lovasz and M Plummer. *Matching Theory*. North-Holland, 1986. Annals of Discrete Mathematics 29.
7. M. Mahajan and K. Varadarajan. A new NC-algorithm for finding a perfect matching in planar and bounded genus graphs. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC)*, pages 351–357, 2000.
8. G Miller and J Naor. Flow in planar graphs with multiple sources and sinks. *SIAM Journal on Computing*, 24:1002–1017, 1995.
9. K Mulmuley, U Vazirani, and V Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–131, 1987.
10. V Vazirani. NC algorithms for computing the number of perfect matchings in  $K_{3,3}$ -free graphs and related problems. *Information and Computation*, 80(2):152–164, 1989.

# Equivalence of Search Capability Among Mobile Guards with Various Visibilities

Jae-Ha Lee<sup>1\*</sup>, Sang-Min Park<sup>2\*\*</sup>, and Kyung-Yong Chwa<sup>3</sup>

<sup>1</sup> Computer Engineering Department, Konkuk University, Korea.  
jaehalee@konkuk.ac.kr

<sup>2</sup> Department of Computer Science, Stanford University, USA.  
smpark@stanford.edu

<sup>3</sup> Department of Computer Science, Korea Advanced Institute of Science and Technology, Korea.  
kychwa@jupiter.kaist.ac.kr

**Abstract.** Given a polygonal region with  $n$  vertices, a group of searchers with vision are trying to find an intruder inside the region. Can the searchers find the intruder or can the intruder evade searchers' detection for ever? It is likely that the answer depends on the visibility of the searchers, but we present quite a general result against it. We assume that the searchers always form a simple polygonal chain within the polygon such that the first searcher moves along the boundary of the polygon and any two consecutive searchers along the chain are always mutually visible. Two types of visibility of the searchers are considered: on the one extreme every searcher has  $360^\circ$  vision – called an  $\infty$ -searcher; on the other extreme every searcher has one-ray vision – called a 1-searcher. We show that if any polygon is searchable by a chain of  $\infty$ -searchers it is also searchable by a chain of 1-searchers consisting of the same number of searchers as the  $\infty$ -searchers. Our proof uses simple simulation techniques. The proof is also interesting from an algorithmic point of view because it allows an  $O(n^2)$ -time algorithm for finding the minimum number of 1-searchers (and thus  $\infty$ -searchers) required to search a polygon [9]. No polynomial-time algorithm for a chain of multiple  $\infty$ -searchers was known before, even for a chain of two  $\infty$ -searchers.

## 1 Introduction

The visibility-based pursuit-evasion problem is that of planning the motion of one or more searchers in a polygonal environment to eventually see an intruder that is capable of moving arbitrarily fast [14,5,2,15,4,8,10,6]. A searcher *finds* an intruder if the intruder is within the range of the searcher's vision sensor

---

\* This research was supported by grant No. R08-2003-000-10569-0(2003) from the Basic Research Program of the Korea Science and Engineering Foundation and by University IT Research Center Project.

\*\* Supported by the Post-doctoral Fellowship Program of Korea Science and Engineering Foundation.

at any moment. This problem can model many practical applications such as search for an intruder in a house, rescue of a victim in a dangerous area and surveillance with autonomous mobile robots. The motion plan calculated could be used by robots or human searchers. Some pursuit-evasion problems take the intruder(evader)'s strategy into account, which makes searching a *game* between two parties [12,1,13].

In general, the goal of the visibility-based pursuit-evasion problem is to compute a schedule for the searchers so that any intruder will eventually be detected by the searchers regardless of the unpredictable trajectory of the intruder, which is, however, not always achievable. A polygon is said to be *searchable* by given searchers if there exists such a schedule. Whether a polygon is searchable or not depends on many factors such as the shape of the polygon, the number of searchers, and the visibility that the searchers have. The visibility of a searcher can be defined by the number of flashlights. The  $k$ -searcher has  $k$  flashlights whose visibility is limited to  $k$  rays emanating from its position, where the directions of the rays can be changed continuously at bounded angular rotation speed. One can envisage a laser beam as an example of a flashlight, which can detect an intruder crossing it. The searcher with the omnidirectional vision that can see in all directions simultaneously is called an  $\infty$ -searcher.

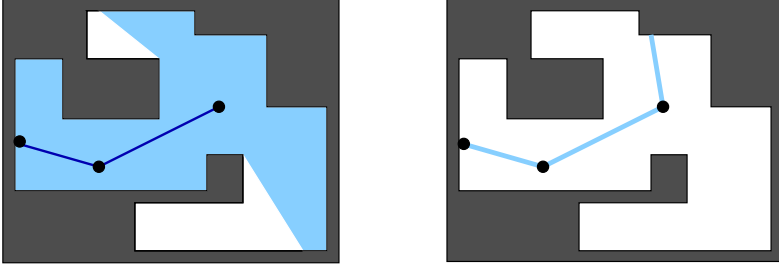
One of the most interesting results given in the previous work is that any polygon (and its variants) searchable by an  $\infty$ -searcher is also searchable by a 2-searcher [2,10,11]. This means an  $\infty$ -searcher and a 2-searcher have the same search capability. These results have crucial impact on algorithms by making the polynomial-time algorithms developed for a 2-searcher complete for an  $\infty$ -searcher. Solely for an  $\infty$ -searcher, no polynomial-time algorithm was developed (refer to [4]).

Most results on the equivalence of search capability rely on characterizations of the class of searchable polygons (or polygon's variants). This implies not only that the proof is quite complicated but also that a proof for one setting cannot be easily adapted for another setting. More specifically, for each search environment - a polygon or its variants, rather different arguments were used for characterizing the class of searchable ones.

Recently, Suzuki *et al.* [16] presented a new approach without resorting to characterizations of searchable polygons. They considered a single searcher moving along the boundary and showed that an  $\infty$ -searcher and a 1-searcher have the same search capability. Their proof constructs a search diagram of an  $\infty$ -searcher and shows that a 1-searcher can do the same job using topological arguments. With the same framework, they also showed that any polygon searchable by an  $\infty$ -searcher with some restrictions is also searchable by a 2-searcher. The latter result was independently shown by the authors of the present paper [7].

In this paper we study a more general framework and present a simple proof on the equivalence of search capability of *multiple* searchers. We consider a group of  $m$  searchers which form a chain. Let  $s_1, s_2, \dots, s_m$  denote  $m$  searchers such that  $s_1$  moves along the boundary of the polygon and that  $s_i$  and  $s_{i+1}$  are always mutually visible for  $1 \leq i < m$ . We call it an  $m$ -chain. We consider

two types of visibility of the searchers: on the one extreme, the searchers have the omnidirectional vision, so the “*capture region*” is the union of the visibility polygons defined by the positions of the searchers. On the other extreme, each searcher has only one flashlight (that is, a 1-searcher) such that  $s_i$ ’s flashlight must always illuminate  $s_{i+1}$  during the search for  $1 \leq i < m$  and only the last searcher  $s_m$  can control its flashlight arbitrarily; in this case, the “*capture region*” is the union of the line segments between  $s_i$  and  $s_{i+1}$  for  $1 \leq i < m$  and the flashlight of  $s_m$ .



**Fig. 1.** Examples: 3-chain of  $\infty$ -searchers and 3-chain of 1-searchers.

We will show that if a polygon is searchable by an  $m$ -chain of  $\infty$ -searchers it is also searchable by an  $m$ -chain of 1-searchers. To the knowledge of the authors, search equivalence has never been shown for multiple searchers so far. Our proof uses a *direct simulation*; we transform a search schedule of an  $m$ -chain of  $\infty$ -searchers into that of 1-searchers. The main idea is very simple and it appears to be applied to different settings.

Our model is motivated by some previous results. LaValle *et al.* [6] considered the case that the 1-searcher moves along the boundary of the polygon and they presented an  $O(n^2)$ -time algorithm for finding a search schedule of the 1-searcher. Their model corresponds to the special case of  $m = 1$  in our model. Efrat *et al.* [3] considered a model similar to ours. In their model, not only the first searcher but also the last searcher in the chain should move along the boundary and the visible region is the union of line segments between two consecutive searchers. They presented an  $O(n^3)$ -time exact algorithm for finding the minimum number of searchers required to sweep a polygon and an  $O(n \log n)$ -time approximation algorithm with additive constant for finding the minimum number of searchers required to search a polygon. Lee *et al.* [9] improved the time complexity to  $O(n^2)$  and these algorithms can easily be modified to deal with the chain of 1-searchers studied in the present paper.

Without the chain restriction, the problem of finding the minimum number of  $\infty$ -searchers required to search a polygon is in NP-hard [4] and no polynomial-time algorithm is known even for two  $\infty$ -searchers. Some researchers studied the art-gallery-style bound for multiple 1-searchers [15].

In applications such as robot surveillance systems, one ray vision can be easily implemented by one laser beam (refer to [6]). Our equivalence proof shows that

in quite a natural setting - a chain of searchers moving inside a polygonal region, this simple implementation is as powerful as the omnidirectional vision whose implementation appears to be very expensive.

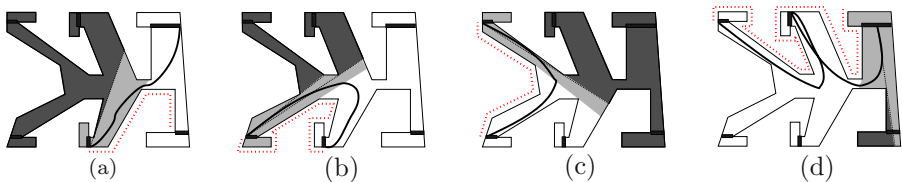
## 2 Definitions and Notations

Let  $\partial\mathcal{P}$  denote the boundary of a simple polygon  $\mathcal{P}$ . The searcher and the intruder are modeled as points that move continuously within  $\mathcal{P}$ . Let  $\phi(t)$  denote the position of the intruder at time  $t \geq 0$ . It is assumed that  $\phi : [0, \infty) \rightarrow \mathcal{P}$  is a continuous function and the intruder is *unpredictable* in that he is capable of moving arbitrarily fast and his path  $\phi$  is unknown to the searcher.

Two points  $p$  and  $q$  are *visible* from each other if the segment  $\overline{pq}$  does not intersect the exterior of  $\mathcal{P}$ . For a point  $q \in \mathcal{P}$ , let  $Vis(q)$  denote the set of all points in  $\mathcal{P}$  that are visible from  $q$ .

Let  $\mathcal{S} = \langle s_1, s_2, \dots, s_m \rangle$  denote a chain of  $m$  searchers. For a searcher  $s_i \in \mathcal{S}$ , let  $\gamma_i(t)$  denote the position of the searcher  $s_i$  at time  $t$ ; we assume that  $\gamma_i(t) : [0, \infty) \rightarrow \mathcal{P}$  is a continuous path. A configuration of  $\mathcal{S}$  at time  $t$ , denoted  $\Gamma(t)$ , is the set of points  $\{\gamma_i(t) | 1 \leq i \leq m\}$ . We say that  $\Gamma(t)$  is *legal* if  $\gamma_1(t)$  lies on  $\partial\mathcal{P}$  and  $\gamma_i(t)$  and  $\gamma_{i+1}(t)$  are mutually visible at  $t$  for  $1 \leq i < m$ . If  $\Gamma(t)$  is legal during the search, then a set of searchers configured by  $\Gamma(t)$  is called an *m-chain* of searchers.

The  $\infty$ -searcher has the omnidirectional vision. Thus at time  $t$ , the  $m$ -chain of  $\infty$ -searchers illuminates all points in the union of  $Vis(\gamma_i(t))$  for all  $i$ . The  $k$ -searcher has  $k$ -flashlights each of which illuminates the points along one ray and can rotate at some bounded speed. Formally, the schedule of the  $k$ -searcher  $s_i$  is a  $(k+1)$ -tuple  $(\gamma_i, \tau_1, \dots, \tau_k)$  such that  $\tau_j : [0, \infty) \rightarrow [0, 2\pi]$  is a continuous function denoting the direction of the  $j$ -th ray, which is a maximal line segment in  $\mathcal{P}$  emanating from  $\gamma_i(t)$ . We define the first boundary point hit by a ray  $f$  as *shot*( $f$ ). The  $m$ -chain of 1-searchers illuminates the points in the union of the segment  $\overline{\gamma_i(t)\gamma_{i+1}(t)}$  for  $1 \leq i < m$  and the segment  $\overline{\gamma_m(t)\text{shot}(f(t))}$  for the flashlight  $f(t)$  of  $s_m$  at  $t$ .



**Fig. 2.** Snapshots of a search path of the 2-chain  $\langle s_1, s_2 \rangle$  of  $\infty$ -searchers. The solid curve represents the path of  $s_2$  and the dotted curve along  $\partial\mathcal{P}$  denotes the path of  $s_1$ .

During a search, a point  $p \in \mathcal{P}$  is said to be *contaminated* at time  $t (\geq 0)$  if there exists a continuous function  $\phi$  such that  $\phi(t) = p$  and that  $\phi(t')$  is not



illuminated at any  $t' \in [0, t]$ . A point that is not contaminated is said to be *clear*. A region  $R \subseteq \mathcal{P}$  is contaminated if it contains a contaminated point; otherwise, it is clear. In the rest of the paper, dark gray indicates contaminated regions and white means clear regions in the illustrations. If  $\mathcal{P}$  is cleared at some time  $t$  by an  $m$ -chain of  $\infty$ -searchers moving along  $(\gamma_1, \gamma_2, \dots, \gamma_m)$ , then the set of paths  $(\gamma_1, \gamma_2, \dots, \gamma_m)$  is called a *search path*. A schedule for an  $m$ -chain of  $k$ -searchers ( $k \geq 1$ ) is called a *search schedule* if  $\mathcal{P}$  is cleared at some time  $t$  during the execution of it. Figure 2 depicts a search path of the 2-chain of  $\infty$ -searchers.

For points  $p, q \in \partial\mathcal{P}$ , let  $[p, q]$  denote the connected boundary chain from  $p$  to  $q$  in the clockwise direction. As usual,  $(p, q)$  denotes the *open* chain  $[p, q] \setminus \{p, q\}$ .

### 3 Sketch of Proof

Suppose that a polygon  $\mathcal{P}$  is searchable by an  $m$ -chain of  $\infty$ -searchers. Let  $S_\infty = (\gamma_1, \gamma_2, \dots, \gamma_m)$  denote a search path for  $\mathcal{P}$ . To show that  $\mathcal{P}$  is also searchable by an  $m$ -chain of 1-searchers we will go through two phases of transformation: first, we will transform  $S_\infty$  into a search schedule of an  $m$ -chain of  $k$ -searchers, called  $S_k$ , where  $k = O(n)$ , and then  $S_k$  will be transformed into a search schedule  $S_1$  of an  $m$ -chain of 1-searchers. A basic idea is to simulate the search process using the smaller number of flashlights. More detailed outlines follow.

During the search by  $S_\infty$ ,  $\mathcal{P}$  is partitioned into some clear regions and some contaminated regions. Thus the boundary of  $\mathcal{P}$  consists of alternating clear and contaminated chains, so that one clear chain lies between two contaminated chains. For the time being, let us focus our attention on the boundary of  $\mathcal{P}$ . If a boundary chain is clear and maximal by  $S_\infty$  at time  $t$ , we call it an *MC-chain* at  $t$ .

Now we will use an  $m$ -chain of  $k$ -searchers for some  $k = O(n)$  to maintain the endpoints of MC-chains, which keep moving on  $\partial\mathcal{P}$  with time. The endpoints of all MC-chains will be kept track of by some flashlights of the  $k$ -searchers. We call them *effective flashlights*. Each MC-chain is bounded by a pair of effective flashlights and from the viewpoint of the  $k$ -searcher who carries them, one is called the *left flashlight* and the other is the *right flashlight*. During this simulation, MC-chains change with time, thus effective flashlights change accordingly; they may appear, move, become non-effective ones or disappear. Detailed explanation is in Section 4.1.

The second phase is to construct  $S_1$  from the dynamics of effective flashlights of  $k$ -searchers. This will be explained in Section 4.2 for the simple case of  $m = 1$  and in Section 5 for the general case ( $m \geq 2$ ).

Throughout the simulation, we concentrate on the boundary chains, which may imply that we deal only with the intruders moving on the boundary of the polygon. In the following sections, however, we will describe the movements of effective flashlights and the process of their simulation, by which you can see how the inside of the polygon is cleared along with its boundary.

## 4 Transformation with a Single Searcher

In this section, we consider a single searcher only ( $m = 1$ ). This will provide a basis of the later proof for the general case ( $m \geq 2$ ), which is more involved.

Suppose that one  $\infty$ -searcher is searching a polygon from the boundary. First, we simulate it using a  $k$ -searcher, for some  $k = O(n)$ . Let  $\gamma$  denote the search path of the  $\infty$ -searcher and let  $S_k$  denote a schedule of the  $k$ -searcher. We will show how to control the effective flashlights in  $S_k$ , and then we construct a search schedule  $S_1$  of the 1-searcher.

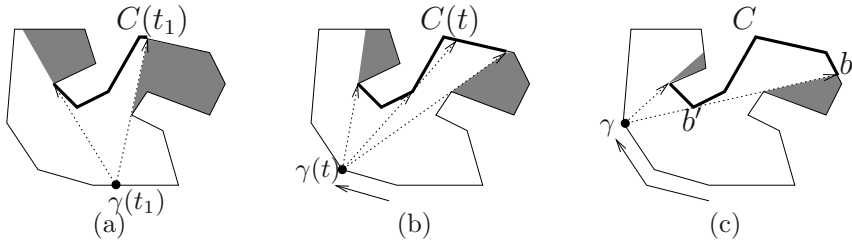
### 4.1 Scheduling a $k$ -Searcher

During the search following  $\gamma$ , MC-chains and contaminated chains appear alternately on  $\partial\mathcal{P}$ . We aim to keep track of endpoints of MC-chains by the flashlights of the  $k$ -searcher so that we can replace the  $\infty$ -searcher by the  $k$ -searcher. Let the path of the  $k$ -searcher be exactly same as that of the  $\infty$ -searcher, that is  $\gamma$ . Let  $\gamma(t)$  be the position of the  $k$ -searcher at time  $t$ . Now we should specify the movements of the flashlights of the  $k$ -searcher.

Let us focus on an MC-chain  $C(t) = [a(t), b(t)]$  at time  $t$ , where  $a(t)$  and  $b(t)$  are functions denoting the endpoints of  $C(t)$ . Now we consider the dynamics of two effective flashlights associated with  $C(t)$ . Suppose the MC-chain appears at  $t_1$  for the first time. Then  $C(t_1)$  is entirely visible from the searcher (see the thick chain in Figure 3a). To simulate  $C(t_1)$ , the  $k$ -searcher turns on two flashlights, aims them at some point in  $C(t_1)$ , and rotates them in opposite directions until the shots of the flashlights reach  $a(t_1)$  and  $b(t_1)$ , respectively. Then these two flashlights are effective ones for  $C$ . When  $a(t)$  and  $b(t)$  move continuously on  $\partial\mathcal{P}$ , the shots of the two effective flashlights move exactly same as they do (see Figure 3b).

Now consider the cases when an endpoint of the MC-chain changes abruptly - not continuously. First suppose the chain  $C$  is extended abruptly, specifically augmented by  $C'$ . We can handle this extension by creating  $C'$  and *merging* it with  $C$ . Without loss of generality, assume that  $C'$  is right to  $C$  (the left case is symmetric). Then the right flashlight of  $C$  will meet the left flashlight of  $C'$ . The  $k$ -searcher turns off two *meeting* flashlights and keeps two remaining ones (left flashlight of  $C$  and right one of  $C'$ ) as the effective flashlights for the extended chain.

Secondly, suppose the chain  $C$  shrinks abruptly. This is the case when a part of the clear chain becomes out of sight suddenly and merged into some contaminated chain. In Figure 3c,  $C$  (the thick chain) gets shortened when the searcher moves from  $\gamma$  clockwise at time  $t$ . The new right endpoint of  $C$  should be the point in  $C(t)$  that is visible from the new position of the searcher, say  $\gamma'$  and first met when we traverse  $\partial\mathcal{P}$  counterclockwise from the old  $b(t)$  (for short  $b$ ). If such a point is not found, this chain  $C$  disappears. Assume otherwise, that is, such a point, say  $b'$ , is found. It is easily seen that three points  $\gamma, b$  and  $b'$  are collinear, because the chain  $(b', b)$  is invisible from  $\gamma'$  and the searcher can move



**Fig. 3.** (a)  $C(t_1)$  is visible from  $\gamma(t_1)$ . (b) The effective flashlight that defines the right endpoint of  $C(t)$  moves continuously. (Note that a part of  $C(t)$  is not visible from  $\gamma(t)$  at the moment.) (c) If the searcher moves any further in a clockwise direction, the right endpoint of  $C$  becomes  $b'$  and  $C$  shrinks.

only at some bounded speed. Since the chain  $(b', b)$  was clear right before  $t$ , we can imagine as if the shot of the flashlight jumps from  $b$  to  $b'$  at  $t$ .

To summarize, each effective flashlight uses two kinds of basic moves besides turn-on and turn-off:

1. *Sweep* : The shot of flashlight moves continuously along the boundary of the polygon (for example, the right endpoint of  $C(t)$  in Figure 3b).
2. *Backward jump* : The shot of flashlight jumps from the current point  $b$  to another boundary point  $b'$  along the line extending the flashlight, where the new shot  $b'$  lies in the MC-chain before the jump, so the MC-chain shrinks (Figure 3c). (“Backward” means that the endpoint of the MC-chain  $C$  retreats and  $C$  loses a part of clear area.)

Notice that a backward jump is defined only when one side of the flashlight is clear. (If it is the left (or right) flashlight, the right (or left) side is clear.)

Any movement that can be represented as a sequence of sweeps and backward jumps is said to be *valid*. The life-cycle of an effective flashlight  $f$  is as follows: it is turned on;  $f$  uses valid movements; it is turned off (at merge or disappearance). The next lemma plays an important role in later proofs.

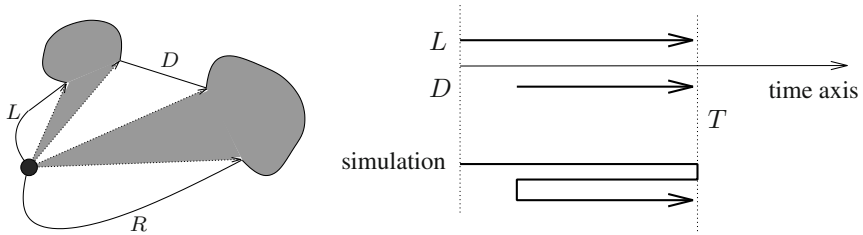
**Lemma 1.** *Suppose a movement of a flashlight is valid with one side of the flashlight clear. Then the reverse execution of this movement is also valid provided the other side is clear.*

**Proof:** The reverse execution of a sweep is also a sweep, so it is valid whichever side is clear. Consider a backward jump with the left side of the flashlight clear (the right side is symmetric). Suppose the shot of a flashlight  $f$  jumped from  $b$  to  $b'$  as in Figure 3c. Now assume that  $f$  aims at  $b'$  currently with the right side of  $f$  clear. The reverse execution would make the shot jump from  $b'$  to  $b$ . Since  $b$  belongs to the clear side before the jump, the new right side of  $f$  is still clear after the jump. Because we can reverse each basic move, the reverse execution of a valid movement is still valid with the opposite side clear.  $\square$

## 4.2 Scheduling a 1-Searcher

In  $S_k$ , the flashlights of the  $k$ -searcher keep track of the endpoints of each MC-chain formed by  $S_\infty$ . For convenience, we are going to maintain two special MC-chains adjacent to the position of the  $\infty$ -searcher,  $\gamma(t)$  - one on the clockwise side, which is denoted by  $L$ , and the other on the counterclockwise side, denoted by  $R$ . It is possible that the neighborhood of  $\gamma(t)$  on both clockwise and counterclockwise sides is clear; in this case, imagine that we simply split the clear chain at  $\gamma(t)$ . Even in the case that  $L = R = \gamma(t)$ , we think as if  $L$  and  $R$  are distinct. Thus  $L$  and  $R$  exist from the beginning of the search and will be merged only when  $L \cup R$  becomes  $\partial\mathcal{P}$ .

The key to the simulation by the 1-searcher is how to handle the merge of MC-chains with only one flashlight. Suppose that according to  $S_\infty$ ,  $L$  is first merged at time  $T$  with some MC-chain  $D$  that is clockwise to  $L$ . So  $L \cap D$  is one point at  $T$ . Assume that  $D$  appeared for the first time at  $t_1$ , where  $t_1 < T$ . Recall that in  $S_k$ ,  $L$  and  $D$  are bounded by effective flashlights of the  $k$ -searcher. The 1-searcher can merge  $L$  and  $D$  in the following way (see Figure 4): The 1-searcher has a single flashlight, say  $F$ . First  $F$  simulates the right flashlight of  $L$  for the time interval  $[0, T]$  in  $S_k$ . Next  $F$  simulates the left flashlight of  $D$  during the time interval  $[t_1, T]$  in  $S_k$  *reversely*; this second simulation proceeds in the time direction opposite to the schedule  $S_k$  of the  $k$ -searcher. This reverse simulation consists of valid movements by Lemma 1. Finally,  $F$  simulates the right flashlight of  $D$  for the time interval  $[t_1, T]$  in  $S_k$ . In this way, the 1-searcher can merge two MC-chains using one flashlight.



**Fig. 4.** Merging chains  $L$  and  $D$  in  $S_1$ .

In the above example,  $D$  may have been created by merge of two (or more) chains, say  $D_1$  and  $D_2$ . Still the flashlight  $F$  can simulate  $D_1$  and  $D_2$  recursively one by one in the same way as for  $L$  and  $D$ . The above argument is summarized as the following lemma.

**Lemma 2.** *Suppose  $L$  is merged with the chain  $D$  during  $S_k$ . Then the 1-searcher can merge  $L$  and  $D$  using the reverse execution.*

The whole schedule  $S_1$  is constructed by applying Lemma 2 repeatedly: the 1-searcher continuously moves along the path  $\gamma$  of the  $k$ -searcher either forward

or backward according to the time direction in which it is being simulated and the flashlight of the 1-searcher merges MC-chains one by one until it amounts to  $\partial\mathcal{P}$ . During the whole simulation, we keep the left side of the flashlight of the 1-searcher being clear.

## 5 Transformation with Multiple Searchers

Suppose that a polygon  $\mathcal{P}$  is searchable by an  $m$ -chain of  $\infty$ -searchers for  $m \geq 2$ . Let  $S_\infty = (\gamma_1, \gamma_2, \dots, \gamma_m)$  denote a search path of the  $m$ -chain of  $\infty$ -searchers. Let  $S_k$  denote the search schedule of an  $m$ -chain of  $k$ -searchers and let  $S_1$  denote that of an  $m$ -chain of 1-searchers. We denote the  $i$ -th  $\infty$ -searcher by  $s_i$ , the  $i$ -th  $k$ -searcher by  $\tilde{s}_i$ , and the  $i$ -th 1-searcher by  $\bar{s}_i$ .

*Constructing  $S_k$ .*  $S_k$  is constructed in the following way: basically, the configuration of the chain of  $k$ -searchers is the exactly same as that of  $\infty$ -searchers. And for each  $i$ , the  $k$ -searcher  $\tilde{s}_i$  simulates the  $\infty$ -searcher  $s_i$  by maintaining effective flashlights for each MC-chain, as shown in Section 4.1. To be exact, an MC-chain here is somewhat different because we are dealing with multiple searchers. Some maximal clear chains may be bounded by two effective flashlights that belong to two different searchers, which implies that such chains cannot be maintained by a single searcher. Thus, we define an  $MC_i$ -chain as a maximal clear chain being maintained by  $s_i$ , which is called a *local* MC-chain. We will use the term *MC-chain* to refer to a globally maximal clear chain. Therefore, an MC-chain is composed of one or more local MC-chains and  $S_k$  is the aggregation of the simulation conducted by  $\tilde{s}_i$  for all  $i \in \{1, 2, \dots, m\}$ .

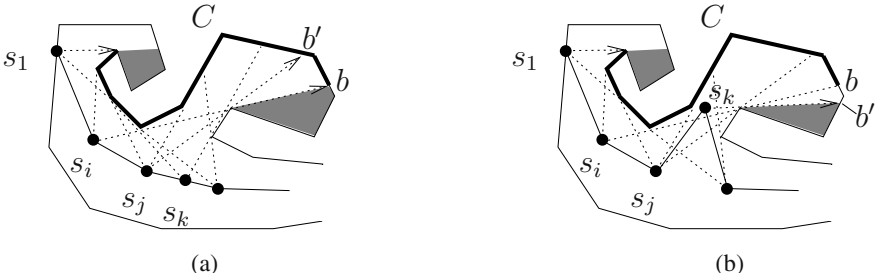
*Interaction among multiple searchers.* Multiple searchers cause more complications because of their interaction. Let us discuss some cases that cannot be handled by the procedure given in Section 4, which are summarize as the following three situations:

1. An MC-chain  $C$  consists of  $MC_i$ -chain and  $MC_j$ -chain for  $i \neq j$ .
2. Two MC-chains are merged and two meeting flashlights belong to two different searchers.
3. An effective flashlight of an MC-chain is switched from  $\tilde{s}_i$ 's to  $\tilde{s}_j$ 's ( $i \neq j$ ).

The first and the second case can be handled by some sorts of merging which will be explained later, and the third one is called *hand-over*. Let us call the most clockwise point of a local or global MC-chain a *cw point*. Because a global MC-chain is the union of some local MC-chains, the role of effective flashlights can be handed over from one searcher to another, according to the dynamic changes of the local MC-chains. Suppose that the cw point of an MC-chain  $C$  is currently defined by a flashlight  $f$  of  $\tilde{s}_i$  (The counterclockwise case is symmetric). If  $f$  performs sweep or backward jump in the counterclockwise direction passing through or over other cw point, then the cw point of  $C$  is switched to the cw point  $x$  that is first encountered by  $f$  moving counterclockwise. If this new cw

point  $x$  is defined by a flashlight of a searcher  $\tilde{s}_j$ , the role of the effective flashlight is handed over from  $\tilde{s}_i$  to  $\tilde{s}_j$ . On the other hand, hand-over can be caused by a searcher  $\tilde{s}_k$  whose flashlight  $f'$  is moving clockwise. If  $f'$  performs sweep in the clockwise direction passing through the flashlight of  $\tilde{s}_i$ ,  $f$ , then  $f'$  becomes the new effective flashlight instead of  $f$ . Figure 5 shows the typical examples of hand-over. In Figure 5a, if the searcher  $s_i$  moves upwards, the cw point of an MC-chain  $C$  which has been defined by the flashlight of  $\tilde{s}_i$  (whose shot is  $b$ ) is now defined by the flashlight of  $\tilde{s}_j$  (whose shot is  $b'$ ). In Figure 5b, as the searcher  $s_k$  moves upwards, its flashlight becomes a new effective flashlight for the chain  $C$ , which induces the extension of  $C$ .

The following observation is important when we construct  $S_1$ : every time that a hand-over occurs, a local MC-chain either changes smoothly or shrinks abruptly (in other words, its flashlights are doing sweep or backward jump). Thus if the right (or left) endpoint of an MC-chain is handed over from the flashlight  $f$  of  $\tilde{s}_i$  to the flashlight  $f'$  of  $\tilde{s}_j$ ,  $f$  and  $f'$  have an intersection and the shot of  $f'$  is to the left (or right) of or equal to that of  $f$  before the hand-over.



**Fig. 5.** Examples of hand-over.

*Constructing  $S_1$ .* Now we will construct the search schedule of 1-searchers,  $S_1$ . Recall that in  $S_1$ , the flashlight of every searcher should be aimed at the neighboring searcher (on the right side) except  $\bar{s}_m$ . With the flashlight of  $\bar{s}_m$ , we should simulate the dynamics of (global) MC-chains. Roughly speaking, the simulation will be done by following the history of each MC-chain and merging one by one. As in the case that  $m = 1$ , we merge MC-chains one by one in the clockwise direction and the flashlight of the last 1-searcher  $\bar{s}_m$ , denoted by  $F$ , will keep track of each MC-chain. When we simulate a left flashlight of an MC-chain, the simulation is done reversely. Before we show how to manage hand-over, merge and initialization, let us give the configuration of an  $m$ -chain of 1-searchers.

We assign the configuration of an  $m$ -chain of 1-searchers as follows: Let  $\gamma(\tilde{s}_i)$  denote the position of  $\tilde{s}_i$  at a given moment. While  $F$  simulates  $\tilde{s}_i$ 's flashlight for some  $i$ , the 1-searchers are located on a minimum-link path (that is a path with the minimum number of turns)  $\pi$  from  $\gamma(\tilde{s}_1)$  to  $\gamma(\tilde{s}_i)$ . Specifically, if  $\pi$  consists of  $l$  segments, we assume that the location of  $\bar{s}_1$  is the same as that of  $\tilde{s}_1$  and  $\bar{s}_2, \bar{s}_3, \dots, \bar{s}_l$  are located at the turning points of  $\pi$  and the remaining

$m - l$  1-searchers are located at the position of  $\tilde{s}_i$ . The existence of such a minimum-link configuration with the number of turns at most  $(m - 2)$  is implied by the configuration of  $\infty$ -searchers. When processing hand-over and merge, the 1-searchers will change their configuration accordingly.

$F$  simulates hand-over as follows: without loss of generality, we fix our attention to the right endpoint of an MC-chain. Suppose that hand-over occurs from flashlight  $f$  of  $\tilde{s}_i$  to flashlight  $f'$  of  $\tilde{s}_j$ . Right before the simulation of this hand-over, the 1-searcher  $\bar{s}_m$  must lie at the position of  $\tilde{s}_i$ . Now the 1-searcher  $\bar{s}_m$  moves to the position of  $\tilde{s}_j$  along the shortest path from  $\tilde{s}_i$  to  $\tilde{s}_j$ , aiming  $F$  at the intersection point, say  $c$ , of  $f$  and  $f'$ . It is easily seen that  $F$  uses valid movements during the above simulation, since hand-over occurs only when the MC-chain shrinks or does not change at the moment of hand-over and the shot of  $f'$  is to the left of or equal to that of  $f$ . (During this simulation, the chain of 1-searchers is morphed to the *minimum-link* configuration from  $\gamma(\tilde{s}_1)$  to  $\gamma(\tilde{s}_j)$  and continues the simulation. Such a morphing can be done by using the routine described in [9].) In case that hand-over occurs between two flashlights illuminating the same point, its simulation is a special case of the above one. A merge of two MC-chains involving two different searchers can be simulated in a similar way to the hand-over because the shots of corresponding flashlights meet at one point.

Finally, when a new MC-chain  $C$  appears (in initialization),  $C$  should have a valid *history* for simulation. When  $C$  appears, every point of this chain must be visible from some  $\infty$ -searcher. We divide  $C$  into a number of sub-chains such that each sub-chain is entirely visible from one  $\infty$ -searcher. We imagine as if each sub-chain was cleared by two flashlights of one  $k$ -searcher and  $C$  was made by merging these sub-chains. In  $S_1$ , it suffices to simulate this imaginary history.

During the execution of  $S_1$ , we satisfy the following invariant: if we assume that an  $m$ -chain of 1-searchers is oriented from  $\bar{s}_1$  to  $\bar{s}_m$ , the left side of the chain of  $m$  flashlights is always clear. Therefore all intruders, not restricted to the ones on  $\partial\mathcal{P}$ , will be detected by  $S_1$ . Through the whole simulation process, we have the following theorem.

**Theorem 3.** *Any polygon that is searchable by an  $m$ -chain of  $\infty$ -searchers is also searchable by an  $m$ -chain of 1-searchers.*

## References

1. M. Adler, H. Racke, N. Sivasadan, C. Sohler, and B. Vocking. Randomized Pursuit-Evasion in Graphs. In *Proc. of the 29th Int. Colloquium on Automata, Languages and Programming*, pages 901–912, 2002.
2. D. Crass, I. Suzuki, and M. Yamashita. Searching for a mobile intruder in a corridor—the open edge variant of the polygon search problem. *Int. J. of Comp. Geom. and Appl.*, 5(4):397–412, 1995.
3. A. Efrat, L.J. Guibas, S. Har-Peled, D. Lin, J. Mitchell, and T. Murali. Sweeping simple polygons with a chain of guards. In *Proc. of the 16th Symp. on Discrete Algorithm*, pages 927–936, 2000.

4. L.J. Guibas, J.C. Latombe, S.M. Lavalley, D. Lin, and R. Motwani. A visibility-based pursuit-evasion problem. *Int. J. of Comp. Geom. and Appl.*, 9(4):471–493, 1998.
5. C. Icking and R. Klein. The two guards problem. *Int. J. of Comp. Geom. and Appl.*, 2(3):257–285, 1992.
6. S. M. LaValle, B. H. Simov, and G. Slutzki. An algorithm for searching a polygonal region with a flashlight. In *Proc. of the 16th Symp. on Comp. Geom.*, pages 260–269, 2000.
7. Jae-Ha Lee, Sang-Min Park, and Kyung-Yong Chwa. On the polygon-search conjecture. *Technical Report TR-2000-157, CS department, KAIST*, 2000.
8. Jae-Ha Lee, Sang-Min Park, and Kyung-Yong Chwa. Searching a polygonal room with one door by a 1-searcher. *Int. J. of Comp. Geom. and Appl.*, 10(2):201–220, 2000.
9. Jae-Ha Lee, Sang-Min Park, and Kyung-Yong Chwa. Optimization algorithms for sweeping a polygonal region with mobile guards. In *Proc. of the 12th Int. Symp. on Algo. and Computation*, pages 480–492, 2001.
10. Jae-Ha Lee, Sung Yong Shin, and Kyung-Yong Chwa. Visibility-based pursuit-evasion in a polygonal room with a door. In *Proc. of the 15th ACM Symp. on Comp. Geom.*, pages 281–290, 1999.
11. Sang-Min Park, Jae-Ha Lee, and Kyung-Yong Chwa. Visibility-based pursuit-evasion in a polygonal region by a searcher. In *Proc. of the 28th Int. Colloquium on Automata, Languages and Programming*, pages 456–468, 2001.
12. T. D. Parsons. Pursuit-evasion in a graph. In *Theory and Applications of Graphs*, pages 426–441, 1976.
13. Günter Rote. Pursuit-evasion with imprecise target location. In *Proc. of the 14th ACM-SIAM Symp. on Discrete Algorithms*, pages 747–753, 2003.
14. I. Suzuki and M. Yamashita. Searching for a mobile intruder in a polygonal region. *SIAM J. Comp.*, 21(5):863–888, 1992.
15. M. Yamashita, H. Umemoto, I. Suzuki, and T. Kameda. Searching for mobile intruders in a polygonal region by a group of mobile searchers. *Algorithmica*, 31(2):208–236, 2001.
16. M. Yamashita, H. Umemoto, I. Suzuki, and T. Kameda. Searching a polygonal region from the boundary. *Int. J. of Comp. Geom. and Appl.*, 11(5):529–553, 2001.



# Load Balancing in Hypercubic Distributed Hash Tables with Heterogeneous Processors<sup>\*</sup>

Junning Liu and Micah Adler

Department of Computer Science, University of Massachusetts, Amherst, MA  
01003-4610, USA. {liujn,micah}@cs.umass.edu

**Abstract.** There has been a considerable amount of recent research on load balancing for distributed hash tables (DHTs), a fundamental tool in Peer-to-Peer networks. Previous work in this area makes the assumption of homogeneous processors, where each processor has the same power. Here, we study load balancing strategies for a class of DHTs, called hypercubic DHTs, with heterogeneous processors. We assume that each processor has a size, representing its resource capabilities, and our objective is to balance the load density (load divided by size) over the processors in the system. Our main focus is the offline version of this load balancing problem, where all of the processor sizes are known in advance. This reduces to a natural question concerning the construction of binary trees. Our main result is an efficient algorithm for this problem. The algorithm is simple to describe, but proving that it does in fact solve our binary tree construction problem is not so simple. We also give upper and lower bounds on the competitive ratio of the online version of the problem.

## 1 Introduction

Structured Peer-to-Peer(P2P) systems have been increasingly recognized as the next generation application mode of the Internet. Most of them, such as CAN [17], Chord [19], Pastry [18] and TAPESTRY [21] etc. are *distributed hash tables* (DHTs), which determine where to store a data item by hashing its name to a prespecified address space, and this address space is partitioned across the processors of the P2P system in a manner that allows a specific hash address to be found relatively easily. The aspect of DHT systems that motivates our work is load balancing. This is crucial to a DHT: a major design goal of P2P systems is to provide a scalable distributed system with high availability and small response time.

In this paper, we consider load balancing in a DHT consisting of heterogeneous processors. While most previous work has analyzed DHTs under the assumption that all processors are identical, real DHTs consist of processors with significantly different characteristics in terms of computational power, bandwidth availability, memory, etc. One way to extend results for the homogeneous case to heterogeneous systems is to use *virtual processors*: powerful processors

---

<sup>\*</sup> This work supported by NSF grants EIA-0080119, CCR-0133664, and ITR-0325726.

pretend to be multiple less powerful processors. This approach has the drawback that a processor must maintain a lot of pointers to other processors for each virtual node it represents. Data Migration is another technique dealing with heterogeneous processors. Data migration can be a good approach when data is highly dynamic, but is usually unfavorable when processors are highly dynamic. When processors arrive and leave frequently, the migration process may have a cascading effect of migration [3] that further deteriorates bandwidth usage and data consistency.

This paper tries to balance the load without using virtual servers or migration methods. We study one variant of the DHT, the *hypercubic hash table* (HHT), a variant of CAN [17] described in [2]. The HHT partitions the address space using a full binary tree. The leaves of the tree correspond to the processors of the DHT. We assign a 0 to every left branch in the tree, and a 1 to every right branch. Each processor stores all hash addresses whose prefix matches the string obtained by following the path from the root to the leaf for that processor, and thus the fraction of the address space stored at a processor  $i$  is  $1/2^{l_i}$ , where  $l_i$  is the distance in the tree from  $i$  to the root.

In the hypercubic hash table, the arrival of a new processor  $i$  is handled by *splitting* an existing leaf node  $j$ . To do so, the leaf node for  $j$  is replaced by an internal node with two children  $i$  and  $j$ . The address space is then reallocated as appropriate for this change. Deletions are handled in an analogous manner. [2] analyzed distributed techniques for choosing which node of the tree to split in order to keep the resulting tree as balanced as possible. In particular, the objective was to minimize the ratio between the maximum fraction of the address space stored at any processor, and the minimum fraction of the address space stored at any processor.<sup>1</sup>

For heterogeneous processors, we want to store larger fractions of the address space on some of the processors, and thus we no longer want as balanced a tree as possible. We assume that every processor  $i$  has a single measure of its power, which we call its size, and denote by  $2^{s_i}$  (it will be more convenient for us to use the logarithm of the sizes). Instead of balancing the load across the processors, we now wish to construct a tree that balances the *load density*, where the load density on processor  $i$  is defined to be  $ld_i = L \frac{1}{2^{l_i}} \frac{1}{2^{s_i}} = \frac{L}{2^{s_i+l_i}}$ ,  $L$  is the total load and  $l_i$  is the height (distance from the root) of node  $i$  in the tree. Our goal is to minimize the quantity  $\frac{\max_i ld_i}{\min_i ld_i}$ . This criteria is a natural extension of the load balancing criteria of [2], and characterizes the application's major requirements. An alternative criteria would be to minimize the maximum load density. A drawback to considering only the maximum load density is that it can result in processors with a large size not having a large fraction of the address space: i.e., powerful processors may be underutilized. In fact, our main result actually demonstrates that it is possible to minimize both criteria simultaneously: we

<sup>1</sup> When the number of data items is large compared to the number of processors, and the data items have approximately the same size, this will be a good estimate of the load balance. The case of heterogeneous processors where the number of data items is close to the number of processors has been studied in [5].

describe an algorithm that finds the tree with the minimum load density ratio, but this tree also minimizes the maximum load density.

While the eventual goal of a load balancing technique would be to develop a distributed and online algorithm, it turns out that minimizing the load density ratio with heterogeneous processors is challenging even in a centralized and offline setting. Thus, in this paper we focus on that setting; developing distributed algorithms that utilize the techniques we introduce for the centralized setting is an interesting and important open problem. The centralized problem reduces to the following simple to state algorithmic problem: given a set of nodes  $S = \{p_1, p_2, \dots, p_n\}$  with weights  $s_1, \dots, s_n$ , construct a full binary tree with leaf set  $S$ . Let the *density* of node  $i$  in this tree be  $s_i + l_i$ . The binary tree should minimize the difference between the maximum density of any node and the minimum density of any node. This is a quite natural question concerning the construction of binary trees, and thus we expect that algorithms for this problem will have other applications.

This algorithmic problem is reminiscent of building optimal source codes; for that problem Huffman's algorithm (see [6] for example) provides an optimal solution. In fact, if we use the simpler optimization criteria of minimizing the maximum density, then Golumbic's minimax tree algorithm [11], a variant of Huffman's algorithm, provides the optimal solution. In this variant, when two nodes are merged into a single node, the new node has a size equal to the maximum of the two merged nodes plus one (instead of the sum of the two nodes as in Huffman's algorithm). Similarly, if we must maximize the minimum density, then using the minimum of the two merged nodes plus one gives the optimal solution. What makes our problem more difficult is that we must simultaneously take both the maximum density and the minimum density into account. We point out that there are simple example inputs for which it is not possible to construct a tree that simultaneously minimizes the maximum density and maximizes the minimum density.

Our main result is the introduction and analysis of a polynomial time algorithm, called the *marked-union* algorithm, that, given a set of nodes with integer weights, finds a tree that minimizes the difference between the maximum density and the minimum density. This algorithm is in fact based on the maximin version of Golumbic's algorithm, and as a result also maximizes the minimum density (which corresponds to minimizing the maximum load density on any processor). The difference between our algorithm and Golumbic's is that ours relies on an additional tie breaking rule that determines which nodes to merge. This tie breaking rule is crucial: Golumbic's algorithm without the tie breaking rule can result in a density difference that is considerably worse than the optimal.<sup>2</sup>

The fact that the marked-union algorithm does in fact minimize the density difference is not obvious, and in fact the proof of this is somewhat involved. As

<sup>2</sup> For example, on an input of  $2^n - 1$  nodes of the same size, if ties are broken arbitrarily, then the density difference can be as bad as  $n - 1$ , whereas the optimal for this input is 1. We also point out that while this tie breaking rule does not reduce the maximum load density in the system, it can significant reduce the number of processors that must deal with this maximum load density.

further evidence that minimizing the density difference is not as easy to deal with as the minimax and the maximin optimization criteria, we point out that the marked-union algorithm does not always return the optimal solution for non-integer inputs, and in fact the complexity of that case is an open problem. Nevertheless, the marked-union algorithm does provide a 2-approximation.

We also provide lower and upper bounds on the competitive ratio of the online version of this problem. In this scenario, the algorithm has access to the entire current tree, and, on an arrival, must decide which node to split to achieve as good load balancing as possible.

This paper is organized as follows: In Section 2, we define the problem more formally and present the marked-union algorithm. In Section 3, we prove the optimality of the marked-union algorithm. In Section 4 we describe our results for the competitive analysis of the online algorithm. Finally, in Section 5 we discuss future work. Due to space limits, most of our proofs can be found in our technical report [13].

## 1.1 Related Work

There has been a number designs on how to build a scalable structured P2P system, including CAN [17], HHT [2], Chord [19], Viceroy [14], Pastry [18], Tapestry [21], Distance Halving DHT [15], and Koorde [12]. Most of these DHTs achieve a  $\log n$  ratio of load balancing with high probability, although many also consider schemes for improving this to  $O(1)$ , sometimes at the cost of other performance measures. CAN [17] allows a measure of choice to make the load more balanced. This technique is analyzed in [2] for the closely related HHT; that paper demonstrates that it achieves a constant load balance ratio with high probability. All of the work mentioned above considers only the homogenous scenarios.

Work on heterogenous load balancing has just begun. [16] and [1] provide heuristics for dealing with heterogenous nodes. In [16], migration and deletion of virtual servers is simulated in a system based on Chord. [1] uses the P-Grid Structure. There, each node is assumed to know its own best load and the system reacts accordingly to balance the load. This provides further motivation for our offline algorithm: it demonstrates that determining the optimal allocation for the processors currently in the system is a useful tool for load balancing of heterogenous P2P systems.

Perhaps the closest work to ours is [7], which introduces a protocol and proves that it balances the heterogenous nodes' load with high probability for a variant of the Chord DHT, assuming that a certain update frequency is guaranteed. To the best of our knowledge, there has been no competitive analysis of load balancing problems for structured P2P systems.

There has been a considerable amount of work on Huffman codes and its variants, which is related to our problem of constructing binary trees. [20] showed that if the input is pre-sorted, the running time of Huffman's algorithm can be reduced to be  $O(n)$ . [11] introduced the minimax tree algorithm already mentioned. [10] generalizes the Huffman code to a non-uniform encoding alphabet

with the same goal of minimizing the expected codeword length. They gave a dynamic programming algorithm that finds the optimal solution for some cases, as well as a polynomial time approximation scheme for others.

There is recently some work studying constrained version of optimization problems arising in binary tree construction. [8] studied the problem of restructuring a given binary tree to reduce its height to a given value  $h$ , while at the same time preserving the order of the nodes in the tree and minimizing the displacement of the nodes. [9] considers the problem of constructing a nearly optimal binary search trees with a height constraint.

## 2 Problem Statement and Algorithm

We next provide a more formal description of the problem. Suppose we have  $m$  different possible processor sizes,  $S_m > S_{m-1} > \dots > S_i > \dots > S_1 > 0$ . Each size is an integer, and there are  $n_i > 0$  processors for size  $S_i$ . Define a *solution tree* to be a full binary tree plus a bijection between its leaf nodes and the input processors. Let  $T$  be any solution tree. Denote the depth of a processor  $R$  in a tree  $T$  as  $l_R$ , the size of it as  $s_R$  (the root node has a depth of zero). The *density* of processor  $R$  is  $d_{T,R} = s_R + l_R$ .<sup>3</sup> The *density difference* of  $T$  is the maximum density minus the minimum density. Our goal is to find the *optimal solution tree*  $T^*$  that achieves the minimum density difference  $\text{Minimum}_T(\max_R d_{T,R} - \min_R d_{T,R})$ .

For this problem we design the marked-union algorithm. In order to describe it, we start with some notation. This algorithm will repeatedly merge *nodes*. To start with, the set of input processors form the set of nodes. We refer to these initial nodes as *real nodes*. Each node maintains a minimum density attribute  $d_{\min} = x$ . We sometime refer to a node as  $(x)$ . The real node for a processor has the processor's size as its  $d_{\min}$ .

The algorithm proceeds via a series of *unions*: an operation that consumes two nodes, and produces a *virtual node*. The nodes it consumes can be real or virtual. When a union operation consumes nodes  $(d_{1\min})$  and  $(d_{2\min})$  the new node it generates will be  $(\min(d_{1\min}, d_{2\min}) + 1)$ .

The aspect of our algorithm that makes it unique as a variant of Huffman's algorithm is the concept of a *marked node*. A virtual node is labeled as marked when it is formed if either of two conditions hold: (1) the two input nodes that create that node have different values of  $d_{\min}$ , or (2) either of the two input nodes was already marked. The marked-union algorithm will use this marking to break ties; this will lead us to the optimal solution.

We are now ready to describe the algorithm as below:

Note that all the input nodes start as unmarked real nodes and we will prove in Lemma 1 later that there will never be more than one marked node. The algorithm's running time is just  $O(N \log N)$ , where  $N = \sum_{i=1}^m n_i$ . Also, we can use it to return a solution tree: treat each real node as a leaf node, and on any

<sup>3</sup> We will use the density for the offline analysis, but use the load density for the cost function of online competitive analysis. Note their relationship is  $d_{T,R} = \log L - \log ld_{T,R}$

**The Marked-Union Algorithm:**

- (1) **Initialization:**  
Sort the input nodes in decreasing order of size. Construct the working queue using the nodes in this order (from left to right).
- (2) **Processing the nodes**
- (3) While (more than one node remains in the working queue) {
- (4)     Union the rightmost two nodes, resulting in a new node  $V(d)$ ;
- (5)     Insert  $V(d)$  into the working queue in the following order:  
From left to right, nodes decrease according to  $d_{min}$ ; for nodes with the same  $d_{min}$ , marked nodes appear to the left of unmarked nodes, otherwise, ties are broken arbitrarily.
- (6) } end while loop

**Fig. 1.** Marked-Union algorithm.

union operation, the resulting virtual node is an internal node of the tree with pointers to the two nodes it consumed. Our main result will be to show that this tree is an optimal solution tree, which we call the *best tree*.

We also point out that we can also get the maximum density difference of the resulting tree at the same time. To do so, we modify the algorithm by adding a new attribute to any marked node which stores the maximum density of any node in its subtree. If we represent a marked node as  $V(d_{min}, d_{max})$ , modify the union operation as:  $(d_{1min}) \cup (d_{2min}) = (\min(d_{1min}, d_{2min}) + 1, \max(d_{1min}, d_{2min}) + 1)$  when  $d_{1min} \neq d_{2min}$ ; and  $(d) \cup (d_{min}, d_{max}) = (\min(d, d_{min}) + 1, \max(d, d_{max}) + 1)$  when  $d_{1min} = d_{2min} = d$ . As was already mentioned, we do not need to define a union operation for the case where two marked nodes are consumed. By defining union this way, it is easy to see that  $d_{min}$  will always be the minimum density in the subtree rooted at that node, and  $d_{max}$  will always be the maximum density. Thus, the density difference of the final marked node  $d_{max} - d_{min}$  will be the maximum density difference of the tree. If the final node is unmarked, all nodes have the same density.

## 2.1 Marked-Union with the Splitting Restriction

Before we prove the optimality of the tree resulting from this algorithm, we show that this tree can be constructed under the restriction that it is built up using a sequence of splitting operations. In other words, we assume that the nodes must be inserted into the tree one at a time, and each such insertion must be handled by splitting a node of the tree. In order to do so, we first sort the processor by size. We assume that the input processor sequence is  $p_m, p_{m-1}, \dots, p_2, p_1$  with sizes satisfying  $s_m \geq s_{m-1} \geq \dots \geq s_2 \geq s_1$ . We next run the marked-union algorithm and record the resulting depth  $l_i^*$  for each processor  $p_i$ .

We then process the whole input sequence from left to right. For the first node, we use it as the root node of the solution tree. For each subsequent node  $p_j$ , choose the leftmost node  $p_i$  of the already processed input sequence that has

a current depth  $l_i < l_i^*$  in the solution tree thus far. As we demonstrate in the proof of Theorem 1 in [13], such a node must exist. Split the node  $p_i$  and add  $p_j$  as its new sibling. When all processors have been processed, return the final tree. Let  $T^*$  be this tree.

**Theorem 1.** *In the tree  $T^*$ , each processor  $p_i$  will have  $l_i = l_i^*$ .*

### 3 The Optimality of the Marked-Union Algorithm

**Theorem 2.** *The marked-union algorithm returns an optimal solution tree w.r.t. the minimum density difference.*

This section is devoted to proving Theorem 2. We start with a high level overview of this proof. The first step (Section 3.1) is to define the concept of rounds. Very roughly, there is one round for each different size processor that appears in the input, and this round consists of the steps of the algorithm between when we first process nodes of that size and when we first process nodes of the next larger size from the original input. Once we have defined rounds, we prove a number of properties concerning how the virtual nodes in the system evolve from one round to the next.

The second step of the proof (Section 3.2) defines *regular trees*, a class of solution trees that satisfy a natural monotonicity property. There always exists some optimal solution that is a regular tree, and thus we prove a number of properties about the structure of these trees. In particular, we examine sets of subtrees of a regular tree. There is one set of subtrees for each processor size, and these subtrees are formed by examining the lowest depth where that processor size appears in the tree. The third step (Section 3.3) uses the results developed in the first two steps to show that the tree produced by the marked-union algorithm has a density difference that is no larger than any regular tree. We do so via an induction on the rounds of the algorithm, where the virtual nodes present after each round are shown to be at least as good as a corresponding set of subtrees of any regular tree.

#### 3.1 Analysis of Marked-Union Algorithm

Now let us do the first step of the proof for Theorem 2. We first introduce the definition of Rounds for the purpose of analyzing the algorithm. Note that rounds is not a concept used in the algorithm itself.

Let us divide the algorithm into  $m$  rounds, each round contains some consecutive union operations: Round<sub>1</sub>'s start point is the start point of the algorithm. Round <sub>$i$</sub> 's start point is the end point of Round <sub>$i-1$</sub> ; its end point is right before the first union operation that will either consume a real node of size  $S_{i+1}$  or will generate a virtual node with a size bigger than  $S_{i+1}$ . All unions between these two points belongs to Round <sub>$i$</sub> . Round <sub>$m$</sub> 's end point is when we have a single node left as the algorithm halts.

**Lemma 1.** *There can be at most one marked node throughout one run of the algorithm, and the marked node will always be a node with the smallest  $d_{min}$  in the working queue.*

Let  $\Delta_i$  be  $d_{max} - d_{min}$  of the marked node before Round $_{i+1}$  after Round $_i$ , and  $\Delta_i = 0$  if there is no marked node at that time. We will prove in Lemma 2 that after each Round all virtual nodes will have the same  $d_{min}$ . Let  $d_i$  be the  $d_{min}$  of the virtual nodes after Round $_i$ .

**Lemma 2.** *Before Round $_{i+1}$ , after Round $_i$ , we have two possible cases:*

- a) *Single node  $(d_i, d_i + \Delta_i)$ , with  $d_i < S_{i+1}$ ,  $\Delta_i \geq 0$ ;*
- b) *A set of virtual nodes with the same  $d_{min}$ , which are  $k(k \geq 0)$  unmarked virtual nodes  $V(d_i)$  and another marked or unmarked node  $V(d_i, d_i + \Delta_i)$ ,  $d_i = S_{i+1}$ ,  $\Delta_i \geq 0$ .*

**Lemma 3.** *There are two possible cases for  $\Delta_i$ :*

- case a):  $\Delta_i = \max(S_i - d_{i-1}, \Delta_{i-1})$
- case b):  $\Delta_i = 1$ ,  $\Delta_{i-1} = 0$ , and  $d_{i-1} = S_i$

If after Round $_i$ , case b) happens, then we call Round $_i$  a *b-round*, if not, then Round $_i$  is a normal round. Note the first marked node appears at case b) and this case can only happen once: there could be at most one b-round throughout one run of the algorithm.

**Corollary 1.**  $\Delta_i = 0 \Rightarrow \Delta_{i-1} = \Delta_{i-2} = \dots = \Delta_1 = 0$ ,  $d_{j-1} = S_j \forall j \leq i$ .

**Theorem 3 (Gol76).** *The marked-union algorithm maximizes the final  $d_{min}$ .*

### 3.2 Definitions and Conclusions About Regular Trees

Now let us do the second step of the proof of Theorem 2. We first introduce the concept of a special class of solution trees then prove some properties of it. A *regular tree* is a Solution Tree which satisfy that for any two processors, if  $s_i > s_j$  then  $l_i \leq l_j$ .

**Observation 1.** *There exists an optimal solution tree that is a regular tree.*

For any regular tree, let  $l_{i,max}$  be the maximum depth of the leaf nodes with Size  $S_i$ . By the definition of regular tree, it is easy to see  $l_{m,max} \leq l_{m-1,max} \leq \dots \leq l_{2,max} \leq l_{1,max}$ .

Then if we look through across the regular tree at depth  $l_{i,max}$ , the nodes we get are some internal nodes or leaf nodes. For each of the internal nodes  $I_V$ , all its descendants and  $I_V$  form a subtree with  $I_V$  as the root; for each leaf node with a processor size of  $s < S_i$ , it can also be viewed as a subtree of one node, and thus we have a set of subtrees. Define  $k_{i-1}(k_{i-1} \geq 1)$  as the number of these subtrees. Define the set of these subtrees as  $V_{i-1} =$



$\{\nu_{i-1,1}, \nu_{i-1,2}, \dots, \nu_{i-1,j}, \dots, \nu_{i-1,k_{i-1}}\}$ . Furthermore, we define  $k_m = 1$ :  $V_m$  has only one subtree, the regular tree itself.

From its definition, all subtrees of  $V_i$  have the same depth in the regular tree:  $l_{i+1,\max}$  (the depth of a subtree means its root's depth). From the definition of Regular Tree and  $l_{i,\max}$ , we know all the leaf nodes with size  $S_i$  lie in a depth inclusively between  $l_{i+1,\max}$  and  $l_{i,\max}$  in the regular tree, thus  $\nu_{i,1}, \nu_{i,2}, \dots, \nu_{i,j}, \dots, \nu_{i,k_i}$  can be viewed as being constructed from  $\nu_{i-1,1}, \nu_{i-1,2}, \dots, \nu_{i-1,j}, \dots, \nu_{i-1,k_{i-1}}$  plus the  $n_i$  leaf nodes of size  $S_i$ . Notice that in  $V_1$ , we don't have subtrees of  $V_0$  below, they will only be formed by leaf nodes of size  $S_1$ . Also the roots of  $\nu_{i-1,1}, \nu_{i-1,2} \dots \nu_{i-1,j}, \dots \nu_{i-1,k_{i-1}}$  all have the same relative depth of  $l_{i,\max} - l_{i+1,\max}$  in its residing subtree in  $V_i$ . This depth is also the max relative depth in  $V_i$  for real nodes with size  $S_i$ , and there is at least one leaf node with size  $S_i$  that lies at such a relative depth in one of the subtrees of  $V_i$ .

Let *relative density* be a node's density with respect to some subtree of the regular tree, i.e., if a node of size  $S_i$  has a depth of  $l_i$  in a regular tree, and the node is also in some subtree with  $l_r$  as its root's depth in the regular tree, then the node's relative density with respect to the subtree is  $(l_i - l_r) + S_i$ . From above we know the set of subtrees in  $V_i$  all have the same root depth at the regular tree, so we can talk about relative density with respect to a set of subtrees. Let  $d_{\min,i}$  be the minimum relative density w.r.t.  $V_i$  of all the leaf nodes that lie in any of the subtrees of  $V_i$ . Let  $d_{\max,i}$  be the maximum relative density. Let  $\Delta_{d_i} = d_{\max,i} - d_{\min,i}$ .

**Claim 1.** *For any regular tree,*

$$\Delta_{d_i} \geq \Delta_{d_{i-1}} \quad (1)$$

$$\Delta_{d_i} \geq |S_i - d_{\min,i-1}| \quad (2)$$

**Corollary 2.** *If after Round $_i$ , we have a single virtual node in our algorithm, then compared with the  $V_i$  of any regular tree on the same input,*

$$d_i \geq d_{\min,i} \quad (3)$$

### 3.3 Comparing Marked-Union's Output to Regular Trees

Now we have prepared enough for the proof of Theorem 2. By Observation 1, there exist at least one regular tree that is optimal. If we can prove no regular trees on the same input can beat our algorithm, then we prove marked-union algorithm is optimal. So the Lemma below is what we need to prove.

**Lemma 4.** *For any given input, let  $\min(\Delta_{d_i})$  be the minimum  $\Delta_{d_i}$  of all regular trees for the same input. For any Round $_i$ , if it is the last round or it is not the  $b$ -round, then  $\min(\Delta_{d_i}) \geq \Delta_i$ .*

Lemma 4 (proof in [13]) is the core part of the whole optimality proof. Combined with Observation 1, it shows that the marked-union algorithm's output will be no worse than any regular tree in terms of density difference, since there exist at least one optimal solution tree which is also a regular tree. We have proved that the marked-union algorithm returns an optimal tree, or it is an optimal algorithm that minimize the solution tree's density difference.

## 4 Competitive Analysis

The marked-union algorithm is designed for the offline case, where we know the entire set of available processors before any decisions need to be made. We here consider the online case, where the algorithm maintains a current tree, and processors arrive sequentially.

### 4.1 The Model

We describe our model in terms of three parties: the adversary (which chooses the input), the online player (the algorithm), and the third party. At the start of the online problem, the third party specifies some common knowledge, known as *problem parameters*, to both the online player and the adversary. In our model, the common knowledge is a set of possible processor sizes. The adversary then serves the online player with a sequence of processor sizes. The only information the online player has prior to an arrival is that the size of that arrival must be in the set of possible sizes given by the third party. The adversary also determines when the input sequence halts; the online player only sees this when it actually happens. Without a restriction on possible sizes (as is imposed by the third party), the adversary can make the performance of any online algorithm arbitrarily bad, and thus this aspect of the model gives us a way to compare the performance of online algorithms. Furthermore, it is reasonable to assume that real DHTs have some knowledge of the arriving processor's sizes, e.g. the range of the sizes.

Since the load density (as opposed to density) represents our real performance metric for DHTs, we use load density in this section. In particular, we use the load density ratio  $2^{d_{\max} - d_{\min}}$  of the final solution tree when the adversary decides it is time to halt. We point out that instead of the cost of the final solution tree, other choices to consider would be a sum of costs or the maximum cost [4]. However, for our case, when the tree is growing, it is not in a stable situation yet. This transition period should be negligible because it is not a typical working state for the system; we are really concerned with the long term effects, for which our model is more appropriate.

### 4.2 Bounds for Competitive Ratios

For minimization optimization problems, the competitive ratio [4] for an online algorithm is defined as the worst case ratio, over all allowed input sequences, between the cost incurred by an online algorithm and the cost by an optimal offline algorithm. As was already mentioned, we use the original load density ratio as the cost function. Since the node density we used earlier is the log of a node's load density, the log value of competitive ratio is then the difference of density differences between the online algorithm and the optimal offline algorithm. Sometimes we use the log value of the real competitive ratio for convenience. Define the set of input processor sizes which the third party specifies as  $\mathfrak{R} = \{S_i | 1 \leq i \leq m\}$ , with  $S_m > S_{m-1} > \dots > S_2 > S_1$ . Define  $\Delta S = S_m - S_1$ , i.e., the largest size

difference for the processors specified by the third party. Since the case where the third party only specifies a single sized processor is very easy to analyze, we only consider the heterogeneous case where  $m > 1$ . Define  $S_{max} = S_m$  as the largest processor's size,  $S_{min} = S_1$  as the smallest processor's size. We first provide the lower bound for the competitive ratio.

**Theorem 4.** *For any deterministic online algorithm for this load balancing problem, the competitive ratio is at least  $2^{\Delta S}$ .*

We point out that the proof in [13] for this is made more complicated by the fact that it is described for an adversary that has a further restriction. In particular, if we further limit the adversary that it must introduce at least one processor of each size given by the third party, then the bound still holds. This demonstrates that the lower bound really is a function of the difference between the largest processor and the smallest processor. Of course, any lower bound for this restricted scenario still holds for the unrestricted scenario.

We next point out that a very simple algorithm can almost match this lower bound. In particular, the lower and upper bounds are within a factor of 2.

**Theorem 5.** *There is an online algorithm with a competitive ratio of  $2^{\Delta S+1}$  for this load balancing problem.*

From this we also know an upper bound for the best competitive ratio any online algorithm could have.

## 5 Future Work

Since our marked-union algorithm efficiently solves the centralized offline optimization problem, the most important open problem is to design a distributed online algorithm. It is possible that algorithms based on ideas from the marked-union algorithm may provide some direction. Although the online lower bound is somewhat discouraging, it is possible that one can do better using randomization. Also, it seems likely that stochastic models of input sequences are easier to deal with. Finally, looking at the case of multiple, independent measures of a processor's power looks like an interesting but challenging question to pursue.

## References

1. K. Aberer, A. Datta, and M. Hauswirth. The quest for balancing peer load in structured peer-to-peer systems, 2003.
2. Micah Adler, Eran Halperin, Richard M. Karp, and Vijay V. Vazirani. A stochastic process on the hypercube with applications to peer-to-peer networks. In *Proceedings of the thirty-fifth ACM symposium on Theory of computing*, pages 575–584. ACM Press, 2003.
3. Masaru Kitsuregawa Anirban Mondal, Kazuo Goda. Effective load balancing of peer-to-peer systems. In *IEICE workshop on Data engineering*, number 2-B-01, 2003.

4. Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, 1998.
5. John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables.
6. Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. John Wiley & Sons, New York, NY, USA, 1991.
7. Matthias Ruhl David R. Karger. New algorithms for load balancing in peer-to-peer systems. Technical Report LCS-TR-911, UC Berkeley, July 2003.
8. William S. Evans and David G. Kirkpatrick. Restructuring ordered binary trees. In *Symposium on Discrete Algorithms*, pages 477–486, 2000.
9. Gaggie. New ways to construct binary search trees. In *ISAAC: 14th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms)*, Organized by Special Interest Group on Algorithms (SIGAL) of the Information Processing Society of Japan (IPSJ) and the Technical Group on Theoretical Foundation of Computing of the Institute of Electronics, Information and Communication Engineers (IEICE)), 2003.
10. Mordecai J. Golin, Claire Kenyon, and Neal E. Young. Huffman coding with unequal letter costs (extended abstract).
11. M. Golumbic. Combinatorial merging. *IEEE Transactions on Computers*, 24:1164–1167, 1976.
12. M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table.
13. Junning Liu and Micah Adler. Marked-union algorithm for load balancing in hypercubic distributed hash tables with heterogeneous processors. Technical Report University of Massachusetts, Amherst Computer Science Technical Report TR04-43, June 2004.
14. Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.
15. Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach, 2002.
16. Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in structured p2p systems, 2003.
17. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.
18. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.
19. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.
20. J. van Leeuwen. On the construction of Huffman trees. In *Proceedings of the 3rd International Colloquium on Automata, Languages, and Programming*, pages 382–410, 1976.
21. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

# On the Stability of Multiple Partner Stable Marriages with Ties

Varun S. Malhotra

Stanford University, Stanford CA 94305, USA,  
vsagar@stanford.edu

**Abstract.** We consider the generalized version of the stable marriage problem where each man and woman's preference list may have ties. Furthermore, each man and woman wishes to be matched to as many of acceptable partners as possible, up to his or her specified quota. Many-to-many version of the stable marriage problem has wide applications in matching retailers and shopkeepers in *e-marketplaces*. We investigate different forms of stability in this context and describe an algorithm to find *strongly stable* matchings (if one exists) in the context of multiple partner stable marriage problem with ties. In the context of Hospital-Residents problem for which only the resident-oriented algorithm for finding a strongly stable matching is known, this algorithm gives a hospital-oriented version (for the same) as well. Furthermore, in any instance of many-to-many stable marriage problem with ties, we show that the set of strongly stable matchings forms a distributive lattice. The results in this paper extend those already known for the one-to-one version and many-to-one version (Hospitals-Residents problem) of the problem.

## 1 Introduction

The stable assignment problem, first described by Gale and Shapley [3] as the stable marriage problem, involves an equal number of men and women each seeking one partner of the opposite sex. Each person ranks all members of the opposite sex in strict order of preference. A matching is defined to be stable if no man and woman who are not matched to each other both prefer each other to their current partners. Gale and Shapley showed the existence of at least one stable matching for any instance of the problem by giving an algorithm for finding it [3]. An introductory discussion of the problem is given by Polya et al. [6], and an elaborate study is presented by Knuth [11]. Variants of this problem have been studied by Gusfield and Irving [5] amongst others, including cases where the orderings are over partial lists or contain ties. It is known that a stable matching can be found in each of these cases individually in polynomial time (Gale and Sotomayor [4], Gusfield and Irving [5]). However, in the case of simultaneous occurrence of incomplete lists and ties, the problem becomes NP-hard (Iwama et al. [10]). Baiou and Balinski [1] amongst others studied the many-to-many version of the stable marriage problem.

McVitie and Wilson [14] pointed out that the algorithm by Gale and Shapley [3], in which men propose to women, generates a *male-optimal* solution in which every man gets the best partner he can in any possible stable matching and every woman gets the worst partner she can in any stable matching. They suggested an egalitarian measure of optimality under which sum of the ranks of partners for all men and women was to be minimized. Irving et al. [7] provided an efficient algorithm to find a stable matching satisfying the optimality criterion of McVitie and Wilson [14]. This problem was extended for the general case of multiple partner stable marriage problem in [2].

Further research into different versions of stable marriage problem with ties in the preference lists gave rise to the notions of *weak stability*, *super stability* and *strong stability* [8]. In the context of the original one-to-one stable marriage problem, weak stability means that there is no couple  $(m, f)$ , each of whom strictly prefers the other over her or his current match. It was shown in [10] that finding the largest weakly stable matching in an instance of stable marriage problem with ties preference lists is NP-Hard. Strong stability is more restrictive in the sense that a matching  $\mathcal{M}$  is strongly stable if there is no couple  $(x, y)$  such that  $x$  strictly prefers  $y$  to his/her partner in  $\mathcal{M}$ , and  $y$  either strictly prefers  $x$  to her/his partner in  $\mathcal{M}$  or is indifferent between them. Finally, a matching  $\mathcal{M}$  is super-stable if there is no couple  $(x, y)$ , each of whom either strictly prefers the other to his/her partner in  $\mathcal{M}$  or is indifferent between them. Polynomial time algorithms were given for finding strongly stable matchings (if one exists) in the context of one-to-one stable marriage problem [12] and many-to-one stable marriage problem [9] (Hospital-Residents problem).

Our work extends the recent results of Irving et al. [9] and Manlove [12] to the many-to-many version of the stable marriage problem with ties, where each man or woman may have multiple partners and answers some of the questions posed in [9].

The need for such an algorithm arises in the context of *e-marketplaces* where in shopkeepers and retailers are matched based on their preferences and the number of parties they wish to be matched to, which is a function of the availability of resources or time. After the matching is done, any party can shop from or sell to a restrictive and more prospective set of candidates, leading to more chances of an agreement. In such a case, finding a *strongly stable* match is extremely desirable. We seek to investigate this type of stability in the multiple partner version of the stable marriage problem with ties (MMSMPT).

## 2 Multiple Partner Stable Marriage Model

Let  $M = \{m_1, \dots, m_{|M|}\}$  and  $F = \{f_1, \dots, f_{|F|}\}$  respectively denote the sets of  $|M|$  males and  $|F|$  females. Every person has a preference order over those

members of the opposite sex that he or she considers acceptable. There may be ties indicating indifference in the preference lists of both males and females. Let  $L_m$  be the preference list of male  $m$  with the most preferred set of partner(s) occurring ahead of others in the list. Similarly,  $L_f$  represents the preference ordering of female  $f$ . Incomplete lists are allowed so that  $|L_m| \leq |F|$ ,  $\forall m \in M$  and  $|L_f| \leq |M|$ ,  $\forall f \in F$ . Each person also has a quota on the total number of partners with which he or she may be matched. Furthermore, a person prefers to be matched to a person in its preference list than to be matched to fewer than the number of persons specified by his or her quota. Let  $q_m$  and  $q_f$  denote the respective quotas of male  $m$  and female  $f$ . The multiple partner stable marriage problem can thus be specified by  $P = (M, F, L_M, L_F, Q_M, Q_F)$  where  $L_M$  and  $L_F$  respectively denote the  $|M| \times 1$  and  $|N| \times 1$  vectors of male and female preference lists, and  $Q_M$  and  $Q_F$  represent the  $|M| \times 1$  and  $|N| \times 1$  vectors of male and female quotas respectively.

In an instance of MMSMPT, a male-female pair  $(m, f)$  is considered *feasible* if  $m$  and  $f$  are in each other's preference lists. That is,  $m \in L_f$  and  $f \in L_m$ . A *matching*  $\mathcal{M}$  is defined to be a set of feasible male-female pairs  $\{(m, f)\}$  such that  $\forall m \in M$ ,  $m$  appears in at most  $q_m$  pairs and  $\forall f \in F$ ,  $f$  appears in at most  $q_f$  pairs. A matching  $\mathcal{M}$  in the context of multiple partner stable marriage problem is said to be *stable* if any feasible pair  $(m, f) \notin \mathcal{M}$  implies that at least one of the two ( $m$  or  $f$ ) is matched to its full quota of partners all of whom he or she considers better. This implies that for any stable matching  $\mathcal{M}$ , there cannot be any unmatched feasible pair  $(m, f)$  that can be paired with both  $m$  and  $f$  becoming better off. Let  $\mathcal{M}_m$  denote the set of partners of male  $m$  in the stable matching  $\mathcal{M}$  and  $\mathcal{M}_f$  is defined similarly. For a feasible pair  $(m, f)$  and a given matching  $\mathcal{M}$ , we define a relation  $\prec$  where  $f \prec_m \mathcal{M}_m$  means that either  $|\mathcal{M}_m| < q_m$ , or  $m$  prefers  $f$  to at least one of its matched partners  $\in \mathcal{M}_m$ . Likewise,  $m \prec_f \mathcal{M}_f$  means that either  $|\mathcal{M}_f| < q_f$ , or  $f$  prefers  $m$  to at least one of its matched partners  $\in \mathcal{M}_f$ . A matching  $\mathcal{M}$  in an instance of MMSMPT is *weakly stable* if there is no pair  $(m, f) \notin \mathcal{M}$  such that  $m \prec_f \mathcal{M}_f$  and  $f \prec_m \mathcal{M}_m$ . Since one-to-one version of stable marriage problem with ties is an instance of MMSMPT, where the  $q_m = 1$  and  $q_f = 1$ ,  $\forall m, f$ , therefore, finding the largest weakly stable matching in MMSMPT is NP-Hard as follows from the results in [10].

Before defining *strong stability* and *super-stability*, we define a relation  $\triangleleft$  for a feasible pair  $(m, f)$  where  $f \triangleleft_m \mathcal{M}_m$  means that either  $f \prec_m \mathcal{M}_m$  or  $m$  is indifferent between  $f$  and at least one of the partners in  $\mathcal{M}_m$ .  $m \triangleleft_f \mathcal{M}_f$  is defined likewise.

**Definition 1 (Strong Stability).** A matching  $\mathcal{M}$  is *strongly-stable* if there is no feasible pair,  $(m, f) \notin \mathcal{M}$ , such that either  $f \prec_m \mathcal{M}_m$  and  $m \triangleleft_f \mathcal{M}_f$ , or  $m \prec_f \mathcal{M}_f$  and  $f \triangleleft_m \mathcal{M}_m$ .

**Definition 2 (Super Stability).** A matching  $\mathcal{M}$  is *super-stable* if there is no feasible pair,  $(m, f) \notin \mathcal{M}$ , such that  $m \triangleleft_f \mathcal{M}_f$  and  $f \triangleleft_m \mathcal{M}_m$ .



### 3 Finding a Strongly Stable Matching

In this section, we describe our algorithm for finding a strongly stable matching (if one exists) for an instance of MMSMPT, and prove its correctness. A male  $m$  (or female  $f$ ) such that  $|\mathcal{M}_m| = q_m$  is said to be *fully-matched* in the matching  $\mathcal{M}$ . During the execution of the algorithm, males and females become engaged, and it is possible for a male/female to be temporarily matched (or engaged) to more partners than his/her specified quota. At any stage, a female/male is said to be over-matched or under-matched according as she/he is matched to a number of partners greater than, or less than the specified quota. We describe a female/male as a *marked* entity if at any time during the execution of the algorithm, she/he has been engaged to more than or equal to its specified quota of partners. The algorithm proceeds by deleting pairs that cannot be strongly stable from the preference lists. By the deletion of a pair  $(m, f)$ , we mean the removal of  $f$  and  $m$  from each other's lists, and, if  $m$  and  $f$  are temporarily matched/engaged to each other, the breaking of this engagement. By the head and tail of a preference list at a given point of time, we mean the first and last ties respectively on that list (note that a tie can be of length 1). We say that a female  $f$  (male  $m$ ) is dominated in a male  $m$ 's (female  $f$ 's) preference list if  $m$  (or  $f$ ) prefers to  $f$  (or  $m$ ) at least  $q_m$  females ( $q_f$  males) who are engaged to it. A male  $m$  who is engaged to a female  $f$  is said to be *bound* to  $f$  if  $f$  is not over-matched or  $m$  is not in  $f$ 's tail (or both).

An *engagement graph*  $G$  has a vertex for each male and each female with  $(m, f)$  forming an edge if male  $m$  and female  $f$  are engaged. A *feasible* matching (set of engagements) in the engagement graph is a matching  $\mathcal{M}$  such that if any male  $m$  is bound to more than his/her quota of partners, then  $m$  is matched to  $q_m$  of these partners in  $\mathcal{M}$  subject to the constraint that  $\mathcal{M}$  has maximum possible cardinality. A reduced assignment graph  $G_R$  is formed from an engagement graph as follows. For each  $m$ , for any female  $f$  such that  $m$  is bound to  $f$ , we delete the edge  $(m, f)$  from the graph and reduce the quota of both  $f$  and  $m$  by one each; furthermore, we remove all other edges incident to  $m$  if the quota of  $m$  gets reduced to 0. Each isolated vertex or a vertex whose quota gets reduced to 0 (corresponding to a male or female) is then removed from the graph. For each male  $m$  and female  $f$  in the remaining graph, we denote the revised quota by  $q'_m$  and  $q'_f$  respectively. Given a set  $Z$  of males (females) in  $G_R$ , define the neighborhood of  $Z$ ,  $N(Z)$  to be the set of vertices (corresponding to opposite sex) adjacent in  $G_R$  to a vertex in  $Z$ . The deficiency of  $Z$  is defined by  $\delta(Z) = \sum_{m \in Z} q'_m - \sum_{f \in N(Z)} q'_f$ . If  $Z_1$  and  $Z_2$  are maximally deficient, then so is  $Z_1 \cap Z_2$  which implies that there is a unique minimal set with maximum deficiency which is called the *critical set* [12].

The algorithm 1, (similar to the one described in [9] with some non-trivial changes) begins by assigning each male to be free (i.e. not assigned to any female) and each female  $f$  to be unmarked. Every iterative stage of the algorithm involves



---

**Algorithm STRONG**


---

```

Initialize each person to be available
Assign each female to be unmarked
Repeat{
  while some male  $m$  is under-matched and has a non-empty list
    for each female  $f$  at the head of  $m$ 's list
      engage  $m$  to  $f$ 
      if  $f$  is fully-matched or over-matched
        set  $f$  to be marked
        for each male  $m$  dominated on  $f$ 's list
          delete the pair  $(m, f)$ 
    form the reduced assignment graph
    find the critical set  $Z$  of males
    for each female  $f \in N(Z)$ 
      for each male  $m$  in the tail of  $f$ 's list
        delete the pair  $(m, f)$ 
  } until  $\{Z == \emptyset\}$ 
Let  $G$  be the final engagement graph
Let  $\mathcal{M}$  be a feasible matching in  $G$ 
if (some marked female is not fully-matched in  $\mathcal{M}$ ) or (some unmarked female
has fewer matches in  $\mathcal{M}$  than its degree in  $G$ )
  No strongly stable matching exists
else
  Output the strongly stable matching specified by  $\mathcal{M}$ 

```

---

**Fig. 1.** Algorithm for finding Strongly Stable Matching

each free male in turn being temporarily engaged to the female(s) at the head of his list. If by gaining a new match a female  $f$  becomes fully or over-matched, then she is said to be *marked* and each pair  $(m, f)$ , such that  $m$  is dominated in  $f$ 's list is deleted. This continues until every male is engaged to his quota of females or more females or has an empty list. We find the reduced assignment graph  $G_R$  and the critical set  $Z$  of males. As proved later, no female  $f$  in  $N(Z)$  can be matched to any of the males from those in its tail in any strongly stable matching, so all such pairs are deleted. The iterative step is reactivated and this entire process continues until  $Z$  is empty, which must happen eventually since if  $Z$  is found to be non-empty, then at least one pair is subsequently deleted from the preference lists. Let  $\mathcal{M}$  be any feasible matching in the final engagement graph  $G$ . Then  $\mathcal{M}$  is a strongly stable matching unless either (a) some marked female  $f$  is not full in  $\mathcal{M}$ , or (b) some unmarked female  $f$  has a number of matches in  $\mathcal{M}$  less than its degree in  $G$ ; in cases (a) and (b) no strongly stable matching exists.

**Lemma 1.** *If the pair  $(m, f)$  is deleted during an execution of Algorithm STRONG, then it cannot block any matching output by the algorithm.*

*Proof.* Let  $\mathcal{M}$  be a matching output by Algorithm STRONG comprising of feasible pairs that are never deleted, and suppose that some pair  $(m, f)$  is

deleted during execution of the algorithm. If  $f$  is fully-matched in  $\mathcal{M}$ , then  $f$  strictly prefers her current  $q_f$  partners in  $\mathcal{M}$  to  $m$ . Since the algorithm proceeds to find the female *worst*<sup>1</sup> strongly stable matching so  $(m, f)$  cannot not a blocking pair in this case.

However, if  $f$  is under-matched in  $\mathcal{M}$ . It is clear that, in order for the pair  $(m, f)$  to be deleted by the algorithm,  $f$  must have been marked at some point during the execution of the algorithm; otherwise  $m$  along with the set of partners of  $f$  in  $\mathcal{M}$  would have been bound to  $f$  and would not have been deleted from  $f$ 's list. Moreover if  $f$  is marked, then the algorithm would not have given any strongly stable marriage; hence the claim follows.  $\square$

**Lemma 2.** *Every male who is fully-engaged or over-engaged in the final engagement graph  $G$  must be fully-matched in any feasible matching  $\mathcal{M}$ .*

*Proof.* The result is true by definition for any male bound to at least  $q_m$  females. Consider the other males engaged in  $G$ . Any set  $S$  of them must be collectively adjacent in  $G_R$  to set of females with at least  $\sum_{m \in S} q'_m$  cumulative quota, otherwise one of them is in the critical set  $Z$ , and hence  $Z \neq \emptyset$ . As a consequence of Hall's Theorem, this means that all these males are fully-matched in any maximum cardinality *feasible* matching in  $G_R$ , and hence they must be fully matched in any feasible matching  $\mathcal{M}$ .  $\square$

**Lemma 3.** *In the final assignment graph  $G$ , if a male is bound to more than  $q_m$  females, then the algorithm will report that no strongly stable matching exists.*

*Proof.* Suppose the algorithm reports that matching  $\mathcal{M}$  is strongly stable. Denote by  $F_1$  the set of over-matched females in  $G$ , let  $F_2 = F \setminus F_1$  and let  $d_G(f)$  denote the degree of the vertex  $f$  in  $G$ , i.e., the number of males temporarily engaged to  $f$ . Denote by  $M_1$  the set of males bound to no more than or equal to their specified quota of females in  $G$ , and  $M_2 = M \setminus M_1$ . However, as a consequence of Lemma 2, for males  $m \in M_2$  who are over-engaged or fully-engaged,  $q'_m = q_m$ . We have

$$|\mathcal{M}| = \sum_{m \in M_1} d_G(m) + \sum_{m \in M_2} q'_m \quad (1)$$

Furthermore,

$$|\mathcal{M}| = \sum_{f \in F_1} q_f + \sum_{f \in F_2} d_G(f). \quad (2)$$

If we consider the procedure used for obtaining the reduced assignment graph, then some male who is over-bound can be matched to only  $q_m$  females; and it follows that

$$\sum_{f \in F_1} (q_f - q'_f) + \sum_{f \in F_2} d_G(f) > \sum_{m \in M_1} d_G(m) \quad (3)$$

---

<sup>1</sup> There can be no strongly stable matching wherein any female has better partners than those in the matching found by the algorithm

Combining (1), (2) and (3) gives

$$\sum_{f \in F_1} q'_f < \sum_{m \in M_2} q'_m, \quad (4)$$

which contradicts the fact that the critical set in  $G_R$  is empty, and hence the claim follows.  $\square$

**Lemma 4.** *Algorithm **STRONG** outputs a strongly stable matching (if it gives a matching as output).*

*Proof.* Suppose Algorithm **STRONG** outputs a matching  $\mathcal{M}$ , and that  $(m, f)$  is a blocking pair for  $\mathcal{M}$ . Then  $m$  and  $f$  are acceptable to each other, so that each is on the original preference list of the other. As a consequence of Lemma 1, the pair  $(m, f)$  has not been deleted which implies that one of  $m$  or  $f$  is on the reduced list of the other. Since,  $(m, f)$  is a blocking pair for  $\mathcal{M}$ , and  $f$  is in  $m$ 's reduced preference list, it cannot follow that  $f \prec_m \mathcal{M}_m$ . Hence,  $f$  must be tied with the tail of  $\mathcal{M}_m$ , and therefore,  $m \prec_f \mathcal{M}_f$ . Therefore,  $m$  is bound to  $f$  in the final engagement graph  $G$ .

The fact that  $m$  is bound to  $f$  in  $G$ , but not matched with  $f$  implies that  $m$  is bound to at least  $q_m$  partners other than  $f$  in  $G$ , by the condition for  $\mathcal{M}$  to be a feasible matching. Hence  $m$  is bound to more than  $q_m$  females in  $G$ , and Lemma 3 contradicts the assumption that the algorithm will output any matching  $\mathcal{M}$ .  $\square$

**Lemma 5.** *No strongly stable pair is ever deleted during an execution of Algorithm **STRONG**.*

*Proof.* Suppose that the pair  $(m, f)$  is the first strongly stable pair deleted during some execution of the algorithm, and let  $\mathcal{M}$  be a strongly stable matching in which  $m$  is matched to  $f$  amongst its other partners, if any. There are two cases to consider.

On one hand, suppose  $(m, f)$  is deleted because of  $m$  being dominated in  $f$ 's list. Call the set of males in the tail of  $\mathcal{M}_f$  (at this point),  $M$ . None of the males in  $M$  can be assigned to any other female they are not matched with and they prefer to  $f$  in any strongly stable matching, for otherwise some strongly stable pair must have been deleted before  $(m, f)$  which contradicts our assumption. Moreover, none of the males in  $M$  can be bound to  $f$  as in that case  $(m, f)$  would not be a strongly stable pair. Therefore, in  $\mathcal{M}$ , at least one of the males in  $M$ ,  $m''$  say, cannot be assigned to  $f$ , so that  $f \triangleleft_{m''} \mathcal{M}_{m''}$ . It follows that  $(m'', f)$  blocks  $\mathcal{M}$ , a contradiction.

On the other hand, suppose  $(m, f)$  is deleted because of  $f$  being matched to some other male in the critical set  $Z$  at some point, and  $m$  is in  $f$ 's tail. We refer to the set of lists at that point as the current lists. It is important to

note that after the reduced assignment graph is constructed, the modified lists of females contain set of equally favorable males (tail of the original list). Let  $Z'$  be the set of males in the critical set  $Z$  who are *fully* assigned in feasible matching  $\mathcal{M}$  and are matched to  $q'_m$  females in their current list, and let  $Y'$  be the set of females in  $N(Z)$  who are assigned in  $\mathcal{M}$  to at least one male from the tail of their current list. Then  $f \in Y'$  since  $(m, f)$  gets deleted implying that  $f$  must be matched to some male in the tail of her current list, so  $Y' \neq \emptyset$ . Furthermore, any male  $m'$  in  $Z$  who is engaged to  $f$  must be in  $Z'$ , for otherwise  $(m', f)$  would block  $\mathcal{M}$ , which implies  $Z' \neq \emptyset$ . We now claim that  $N(Z \setminus Z')$  is not contained in  $N(Z) \setminus Y'$ . For, suppose that the containment does hold.

Then,

$$\begin{aligned} \sum_{m \in Z \setminus Z'} q_m - \sum_{f \in N(Z \setminus Z')} q_f &\geq \sum_{m \in Z \setminus Z'} q_m - \sum_{f \in N(Z) \setminus Y'} q_f \\ &= \sum_{m \in Z} q_m - \sum_{f \in N(Z)} q_f - (\sum_{m \in Z'} q_m - \sum_{f \in Y'} q_f). \end{aligned}$$

But  $\sum_{m \in Z'} q_m - \sum_{f \in Y'} q_f \leq 0$ , because every male in  $Z'$  is matched in  $\mathcal{M}$  with a female in  $Y'$ . Hence  $Z \setminus Z'$  has deficiency greater than or equal to that of  $Z$ , contradicting the fact that  $Z$  is the critical set. Thus the claim is established.

Hence there must be a male  $m_1 \in Z \setminus Z'$  and a female  $f_1 \in Y'$  such that  $m_1$  is engaged to  $f_1$  at some point but they are not matched in the final matching. Since  $f_1 \prec_{m_1} \mathcal{M}_{m_1}$  and  $m_1 \triangleleft_{f_1} \mathcal{M}_{f_1}$ , therefore  $(m_1, f_1)$  blocks  $\mathcal{M}$ , and we reach a contradiction.  $\square$

**Lemma 6.** *Let  $\mathcal{M}$  be a feasible matching in the final engagement graph  $G$ . If (a) some unmarked female  $f$  has fewer matches in  $\mathcal{M}$  than the number of her engagements in  $G$ , or (b) some marked female  $f$  is not full in  $\mathcal{M}$ , then no strongly stable matching exists.*

*Proof.* Suppose that condition (a) or (b) is satisfied, and that  $\mathcal{M}'$  is a strongly stable matching for the instance. Every male fully-engaged (or over-engaged) in the final engagement graph  $G$  must be fully matched in  $\mathcal{M}$  (using Lemma 2), and any male engaged to a fewer number in  $G$  than his specified quota, must be matched to less than or equal to the number of his partners in  $G$ . It follows that  $|\mathcal{M}'| \leq |\mathcal{M}|$  where  $|\mathcal{M}|$  is defined to be the sum of the matches of males in  $\mathcal{M}$ .

If (a) or (b) is true,  $|\mathcal{M}_f| < \min(d_G(f), q_f)$  for some female  $f$ , where  $d_G(f)$  is the degree of  $f$  in  $G$ , i.e., the number of males engaged to  $f$  in  $G$ . Hence also for some  $f'$ ,  $|\mathcal{M}'_{f'}| < \min(d_G(f'), q_{f'})$ . So there is a female  $f'$  and a male  $m' \notin \mathcal{M}'_{f'}$  such that (i)  $f'$  is not full in  $\mathcal{M}'$  and (ii)  $m'$  was engaged to  $f'$  in  $G$ . Hence  $(m', f')$  is a blocking pair for  $\mathcal{M}'$ , contradicting that it is a strongly stable matching.  $\square$

Combining these lemmas leads to the following theorem.

**Theorem 1.** *For a given instance of MMSMPT, Algorithm **STRONG** determines whether or not a strongly stable matching exists. If such a matching does exist, all possible executions of the algorithm find one in which every assigned male is matched to as favorable set of females as in any strongly stable matching.*

## 4 Structure of Strongly Stable Marriages

In this section, we closely study the structure of strongly stable matchings in the context of multiple partner stable marriage problem with ties and interestingly, obtain results that are counterparts to the results [13] for one-to-one version of the problem.

**Lemma 7.** *For a given MMSMPT instance, let  $\mathcal{M}$  be the matching obtained by algorithm **STRONG** and let  $\mathcal{M}'$  be any strongly stable matching. If any female  $f$  is under-matched in  $\mathcal{M}'$  then every male matched to  $f$  in  $\mathcal{M}$  is also assigned to  $f$  in  $\mathcal{M}'$ .*

*Proof.* Suppose any male  $m$  that is matched to  $f$  in  $\mathcal{M}$ , but not in  $\mathcal{M}'$ . Then  $(m, f)$  blocks  $\mathcal{M}'$  since  $f$  is under-matched in  $\mathcal{M}'$  and  $f$  cannot be dominated by the set of partners of  $m$  (Theorem 1).  $\square$

Lemma 7 and the fact that algorithm **STRONG** matches every male to the most favorable set of females he can get matched to in any strongly stable matching, lead to the following theorem.

**Theorem 2.** *For a given MMSMPT instance, every male(female) is matched to the same number of females(males) in every strongly stable matching. Furthermore, any person who is under-matched in some strongly stable matching is matched to the same set of matches in every strongly stable matching.*

Moreover, it can be shown that the set of strongly stable marriages for an instance of MMSMPT form a finite distributive lattice. Before proceeding we state some definitions for the sake of clarity of exposition.

**Definition 3.** *Let  $\mathfrak{M}$  be the set of strongly stable matchings for a given instance of MMSMPT. We define an equivalence relation  $\sim$  on  $\mathfrak{M}$  as follows. For any two strongly stable matchings  $\mathcal{M}_1, \mathcal{M}_2 \in \mathfrak{M}$ ,  $\mathcal{M}_1 \sim \mathcal{M}_2$  if and only if each male is indifferent between  $\mathcal{M}_1$  and  $\mathcal{M}_2$ . Denote by  $\mathfrak{C}$  the set of equivalence classes of  $\mathfrak{M}$  under  $\sim$  and denote by  $[\mathcal{M}]$ , the equivalence class containing  $\mathcal{M} \in \mathfrak{M}$ .*

**Definition 4.** *Let  $\mathcal{M}_1$  and  $\mathcal{M}_2$  be two strongly stable matchings. We define a relation  $\preceq$  wherein  $\mathcal{M}_1 \preceq \mathcal{M}_2$  if and only if each male either strictly prefers  $\mathcal{M}_1$  to  $\mathcal{M}_2$  or is indifferent between them.*

**Definition 5.** *Let  $\mathfrak{C}$  be any equivalence class as defined in Definition 3 for a given instance of MMSMPT. Define a partial order  $\leq$  on  $\mathfrak{C}$  as follows: for any two equivalence classes  $[\mathcal{M}_1], [\mathcal{M}_2] \in \mathfrak{C}$ ,  $[\mathcal{M}_1] \leq [\mathcal{M}_2]$  if and only if  $\mathcal{M}_1 \preceq \mathcal{M}_2$ .*

For any two strongly stable matchings  $\mathcal{M}$  and  $\mathcal{M}'$  for a given instance of MMSMPT, let  $In(\mathcal{M}, \mathcal{M}')$  denote the set of men  $m$  such that  $\mathcal{M}_m \sim \mathcal{M}'_m$ , i.e,  $m$  is indifferent between  $\mathcal{M}$  and  $\mathcal{M}'$ .

**Lemma 8.** *Let  $\mathcal{M}$  and  $\mathcal{M}'$  be two strongly stable matchings in a given instance of MMSMPT. Let  $\mathcal{M}^*$  be a set of (male, female) pairs defined as follows: for each male  $m \in In(\mathcal{M}, \mathcal{M}')$ ,  $m$  has in  $\mathcal{M}^*$  the same partner as in  $\mathcal{M}$ , and for each male  $m \notin In(\mathcal{M}, \mathcal{M}')$ ,  $m$  has in  $\mathcal{M}^*$  the better set of his partners in  $\mathcal{M}$  and  $\mathcal{M}'$ . Then,  $\mathcal{M}^*$  is a strongly stable matching.*

*Proof.* Let us assume that  $(m, f)$  is a blocking pair for  $\mathcal{M}^*$ . Without loss of generality we assume that  $m$  prefers  $f$  to at least one of its set of partners,  $\mathcal{M}_m^*$  in  $\mathcal{M}^*$ . Then  $f$  either prefers  $m$  to at least one of the males in  $\mathcal{M}_f^*$  or is indifferent between  $m$  and the least favourable partner in  $\mathcal{M}_f^*$ . Furthermore,  $m$  prefers  $f$  to at least one partner in both  $\mathcal{M}_m$  and  $\mathcal{M}'_m$ . In either case,  $(m, f)$  either blocks  $\mathcal{M}$  or  $\mathcal{M}'$  depending on whether  $\mathcal{M}_f^*$  is same as  $\mathcal{M}_f$  or  $\mathcal{M}'_f$ . Thus, we reach a contradiction as  $\mathcal{M}$  and  $\mathcal{M}'$  are given to be strongly stable.

Otherwise  $m$  could indifferent between  $f$  and the least preferred partner in  $\mathcal{M}_m^*$ . Thus  $f$  must prefer  $m$  to at least one partner in  $\mathcal{M}_f^*$  for  $(m, f)$  to be a blocking pair for  $\mathcal{M}^*$ . By the construction of  $\mathcal{M}^*$ ,  $m$  prefers  $f$  to at least one partner in  $\mathcal{M}_m$  or is indifferent between  $f$  and the least preferred partner in  $\mathcal{M}_m$ , and  $m$  prefers  $f$  to at least one partner in  $\mathcal{M}'_m$  or is indifferent between  $f$  and the least preferred partner in  $\mathcal{M}'_m$ . Once again we reach a contradiction as  $(m, f)$  either blocks  $\mathcal{M}$  or  $\mathcal{M}'$  depending on whether  $\mathcal{M}_f^*$  is same as  $\mathcal{M}_f$  or  $\mathcal{M}'_f$ . Therefore,  $\mathcal{M}^*$  is strongly stable.  $\square$

Proceeding in a similar way, we get the following lemma.

**Lemma 9.** *Let  $\mathcal{M}$  and  $\mathcal{M}'$  be two strongly stable matchings in a given instance of MMSMPT. Let  $\mathcal{M}^*$  be a set of (male, female) pairs defined as follows: for each male  $m \in \text{In}(\mathcal{M}, \mathcal{M}')$ ,  $m$  has in  $\mathcal{M}^*$  the same partner as in  $\mathcal{M}$ , and for each male  $m \notin \text{In}(\mathcal{M}, \mathcal{M}')$ ,  $m$  has in  $\mathcal{M}^*$  the worse set of his partners in  $\mathcal{M}$  and  $\mathcal{M}'$ . Then,  $\mathcal{M}^*$  is a strongly stable matching.*

We define operations  $\wedge$  and  $\vee$  operations as following.  $\mathcal{M} \wedge \mathcal{M}'$  denotes the set of (male, female) pairs in which each male  $m \in \text{In}(\mathcal{M}, \mathcal{M}')$  is matched to the same set of partners as in  $\mathcal{M}$ , and each male  $m \notin \text{In}(\mathcal{M}, \mathcal{M}')$  gets matched to the better set of his partners in  $\mathcal{M}$  and  $\mathcal{M}'$ . Similarly,  $\mathcal{M} \vee \mathcal{M}'$  denotes the set of (male, female) pairs in which each male  $m \in \text{In}(\mathcal{M}, \mathcal{M}')$  is matched to the same set of partners as in  $\mathcal{M}$ , and each male  $m \notin \text{In}(\mathcal{M}, \mathcal{M}')$  gets matched to the worse set of his partners in  $\mathcal{M}$  and  $\mathcal{M}'$ . Using Lemma 8 and Lemma 9,  $\mathcal{M} \wedge \mathcal{M}'$  and  $\mathcal{M} \vee \mathcal{M}'$  are strongly stable matchings for the given MMSMPT instance.

It is important to note that  $\mathcal{M} \wedge \mathcal{M}' \sim \mathcal{M}' \wedge \mathcal{M}$  because of males  $m \in \text{In}(\mathcal{M}, \mathcal{M}')$ . Similar relation holds true for  $\vee$  operation. Lemma 8 and Lemma 9 lead us to the definition of meet and join operations for the equivalence classes. The given theorem follows using similar arguments as in [13].

**Theorem 3.** *Given an instance  $I$  of MMSMPT, and let  $\mathfrak{M}$  be the set of strongly stable matchings in  $I$ . Let  $\mathfrak{C}$  be the set of equivalence classes of  $\mathfrak{M}$  under  $\sim$  and let  $\preceq$  be the partial order on  $\mathfrak{C}$  as defined in Definition 5. Then,  $(\mathfrak{C}, \preceq)$  forms a finite distributive lattice.*

The consequence of this theorem is that it is possible to efficiently enumerate all the stable matchings in the multiple partner stable marriage case (a result known for one-to-one stable marriage version of the problem).

## 5 Concluding Remarks

The many-to-many version of the stable marriage has wide applications in *advanced matchmaking* in e-marketplaces and finding a strongly stable matching in such a scenario makes best sense. In this text, we present an efficient algorithm for finding a strongly stable marriage (if one exists) for an instance of multiple partner stable marriage problem with ties. Furthermore, we showed that the set of strongly stable matchings forms a distributive lattice. By doing so, we generalized some of the results already known for the corresponding one-to-one and many-to-one (hospital-residents problem) versions of the problem.

Very recently, we have been referred to some recent research work [16] on stable matchings wherein the concept of *level-maximal* matchings is used to bring down the running time of the algorithms for finding strongly stable matchings in the one-to-one and many-to-one versions of the stable marriage problems. The structure of many-to-many stable matchings very closely resembles their one-to-one and many-to-one counterparts. As a result, we believe the algorithm in [16] can be extended to the many-to-many stable marriages with multiple partners. We consider the formalism of such an algorithm as one of the directions for our future work.

**Acknowledgements.** We are thankful to Rob Irving for providing us pointers to some recent research [16] on the stable marriage problem. We are also thankful to the anonymous referees whose suggestions greatly helped in improving the presentation of the paper.

## References

1. Baiou, M., Balinski, M., Many-to-many Matching: Stable Polyandrous Polygamy (or Polygamous Polyandry), *Discrete Applied Mathematics*, Vol 101, (2000) 1–12
2. Bansal, V., Agrawal, A., Malhotra, V.S., Stable Marriages with Multiple Partners: Efficient Search for an Optimal Solution, *ICALP 2003*: 527–542
3. Gale, D., Shapley, L.S., College Admissions and the Stability of Marriage, *American Mathematical Monthly*, Vol 69, (1962) 9–15
4. Gale, D., Sotomayor, M., Some Remarks on the Stable Matching Problem, *Discrete Applied Mathematics*, Vol 11, (1985) 223–232
5. Gusfield, D., Irving, R.W., *The Stable Marriage Problem: Structure and Algorithms*, The MIT Press, Cambridge, 1989
6. Polya, G., Tarjan, R.E., Woods, D.R., *Notes on Introductory Combinatorics*, Birkhauser Verlag, Boston, Massachusetts, 1983
7. Irving, R.W., Leather, P., Gusfield, D., An Efficient Algorithm for the “Optimal” Stable Marriage, *Journal of the ACM*, Vol 34(3), (Jul 1987) 532–543
8. Irving, R.W., Stable marriage and indifference, *Discrete Applied Mathematics* 48(3):261–272, 1994
9. Irving, R.W., Manlove, D., Scott, S., Strong Stability in the Hospitals/Residents Problem, *STACS 2003*: 439–450

10. Iwama, K., Manlove, D., Miyazaki, S., Morita, Y., Stable Marriage with Incomplete Lists and Ties, Proceedings of ICALP, (1999) 443–452
11. Knuth, D.E., Mariages Stables, Les Presses de l'Universite de Montreal, Montreal, 1976
12. Manlove, D., Stable marriage with ties and unacceptable partners, Technical Report, Computing Science Department of Glasgow University, 1999.
13. Manlove, D., The structure of stable marriage with indifference, Discrete Applied Mathematics 122(1–3):167–181, 2002.
14. McVitie, D., Wilson, L.B., The Stable Marriage Problem, Commucations of the ACM, Vol 114, (1971) 486–492
15. Spieker, B., The set of super-stable marriages forms a distributive lattice, Discrete Applied Mathematics 58(1), 79–84, 1995.
16. Telikepalli, K., Mehlhorn, K., Dimitrios, M., Katarzyna, P., Strongly Stable Matchings in Time  $O(nm)$  and Extension to the H/R Problem, *to appear in* Symposium on Theoretical Aspects of Computer Science, 2004.



# Flows on Few Paths: Algorithms and Lower Bounds<sup>\*</sup>

Maren Martens and Martin Skutella

Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken,  
Germany, {martens,skutella}@mpi-sb.mpg.de,  
<http://www.mpi-sb.mpg.de/~{martens,skutella}>

**Abstract.** In classical network flow theory, flow being sent from a source to a destination may be split into a large number of chunks traveling on different paths through the network. This effect is undesired or even forbidden in many applications. Kleinberg introduced the unsplittable flow problem where all flow traveling from a source to a destination must be sent on only one path. This is a generalization of the NP-complete edge-disjoint paths problem. In particular, the randomized rounding technique of Raghavan and Thompson can be applied. A generalization of unsplittable flows are  $k$ -splittable flows where the number of paths used by a commodity  $i$  is bounded by a given integer  $k_i$ .

The contribution of this paper is twofold. First, for the unsplittable flow problem, we prove a lower bound of  $\Omega(\log m / \log \log m)$  on the performance of randomized rounding. This result almost matches the best known upper bound of  $O(\log m)$ . To the best of our knowledge, the problem of finding a non-trivial lower bound has so far been open.

In the second part of the paper, we study a new variant of the  $k$ -splittable flow problem with additional constraints on the amount of flow being sent along each path. The motivation for these constraints comes from the following packing and routing problem: A commodity must be shipped using a given number of containers of given sizes. First, one has to make a decision on the fraction of the commodity packed into each container. Then, the containers must be routed through a network whose edges correspond, for example, to ships or trains. Each edge has a capacity bounding the total size or weight of containers which are being routed on it. We present approximation results for two versions of this problem with multiple commodities and the objective to minimize the congestion of the network. The key idea is to reduce the problem under consideration to an unsplittable flow problem while only losing a constant factor in the performance ratio.

---

<sup>\*</sup> Extended abstract. Information on the full version of this paper can be found at the authors' homepages. This work was partially supported by DFG Focus Program 1126, "Algorithmic Aspects of Large and Complex Networks", grant no. SK 58/4-1, and by EU Thematic Network APPOL II, "Approximation and Online Algorithms", grant no. IST-2001-30012.

# 1 Introduction

*Problem definition.* The unsplittable flow problem (UFP) has been introduced by Kleinberg [11]: Given a network with capacities on the arcs and several source-sink pairs (commodities) with associated demand values, route the demand of each commodity on exactly one path leading from its source to its sink without violating arc capacities. For the special case of unit capacities and unit demands, we get the edge-disjoint path problem which is well-known to be NP-complete [8]. Kleinberg [11] introduced the following optimization versions of the unsplittable flow problem. *Minimum congestion:* Find the smallest value  $\alpha \geq 1$  such that there exists an unsplittable flow that violates the capacity of any edge at most by a factor  $\alpha$ . *Minimum number of rounds:* Partition the set of commodities into a minimum number of subsets (rounds) and find a feasible unsplittable flow for each subset. *Maximum routable demand:* Find a feasible unsplittable flow for a subset of demands maximizing the sum of demands in the subset. Here, we are mainly interested in the minimum congestion problem.

A natural generalization of the unsplittable flow problem is the k-splittable flow problem (k-SFP) which has recently been introduced by Baier, Köhler, and Skutella [4]. For each commodity, there is an upper bound on the number of paths that may be used to route its demand. Already for the single-commodity case, the k-SFP is NP-complete. Of course, the optimization versions of the UFP discussed above naturally generalize to the k-SFP.

In this paper we introduce a new variant of the k-splittable flow problem with additional upper bounds on the amount of flow being sent along each path. As already discussed in the abstract, this generalization is motivated by transportation problems where divisible goods have to be shipped through a network using containers of given sizes. Each container being used, must be routed along some path through the network. In particular, the size of the container induces an upper bound on the amount of flow being sent along the chosen path. It is therefore called *path capacity*.

An edge of the network corresponds, for example, to a train or ship which can be used in order to transport containers from the tail to the head node of this edge. Of course, such an edge cannot carry an arbitrary amount of flow. We consider two variants of the problem for which we provide an intuitive motivation:

- i) When loading a ship, the *total weight* of all containers on the ship must not exceed the capacity of the ship. The weight of a container is determined by the actual amount of flow assigned to it. Thus, in the first variant with *weight capacities*, the capacity of an edge bounds the actual amount of flow being sent on paths containing this edge. This is the classical interpretation of edge capacities.
- ii) When loading a train, the *total size* of the containers is usually more important than their total weight. Therefore, in the model with *size capacities*, the capacity of an edge bounds the total path capacity of all paths being routed through that edge. Notice that these capacity constraints are more restrictive than the classical ones in the first model.

We are mainly interested in the corresponding NP-hard optimization problem to minimize the congestion. A precise and formal definition of the problem under consideration is given in Section 2.

*Related results from the literature.* The unsplittable flow problem has been well studied in the literature. Raghavan and Thompson [15,14] introduced a randomized rounding technique which gives an  $O(\log m)$ -approximation algorithm for the problem of minimizing the congestion for the UFP, if the maximum demand is bounded from above by the minimum edge capacity (the *balance condition*)<sup>1</sup>. For any instance of the problem of minimizing the congestion for the UFP their technique searches for an optimal solution of the related (fractional) multicommodity flow problem first and then chooses exactly one of the flow paths from it for each commodity at random, such that each flow path is chosen with probability equal to its value divided by the demand of its commodity. Their procedure even yields a constant factor approximation, if either the minimum edge capacity or the congestion of an optimal fractional routing is at least  $\Omega(\log m)$ .

For the objective of maximizing the sum of satisfied demands while meeting all edge capacity constraints, Baveja and Srinivasan [5] gave an approximation ratio of  $O(\sqrt{m})$ . Azar and Regev [1] gave a strongly polynomial algorithm also with approximation ratio  $O(\sqrt{m})$ . Kolman and Scheideler [13] even gave a strongly polynomial  $O(\sqrt{m})$ -approximation algorithm for the problem without the balance condition. On the other hand, Guruswami, Khanna, Rajaraman, Shepherd, and Yannakakis [9] showed that in the directed case there is no approximation algorithm with performance ratio better than  $O(m^{-\frac{1}{2}+\epsilon})$  for any  $\epsilon > 0$ , unless  $P=NP$ . To get better approximation results one has to incorporate additional graph parameters into the bound. Baveja and Srinivasan [5] developed a rounding technique to convert an arbitrary solution to an LP-relaxation into an unsplittable flow within a factor of  $O(\sqrt{m})$  or  $O(d)$  where  $d$  denotes the length of a longest path in the LP solution. Using a result of Kleinberg and Rubinfeld [10], showing that  $d = \mathcal{O}(\Delta^2 \alpha^{-2} \log^3 n)$  for uniform capacity graphs with some expansion parameter  $\alpha$  and maximal degree  $\Delta$ , one can achieve a better bound. Kolman and Scheideler [13] use a new graph parameter, the “flow number”  $F$  and improve the ratio further to  $O(F)$  for undirected graphs; they show that  $F = \mathcal{O}(\Delta \alpha^{-1} \log n)$ . Kolman and Scheideler also showed that there is no a better approximation in general, unless  $P = NP$ . Recently Chekuri and Khanna [6] studied the uniform capacity UFP for the case that the task is to satisfy as many requests as possible. Using results from [1] or [5] they found an  $O(n^{2/3})$ -approximation algorithm for the problem in undirected graphs, an  $O(n^{4/5})$ -approximation algorithm for the problem in directed graphs, and an  $O(\sqrt{n} \log n)$ -approximation algorithm for the problem in acyclic graphs. If all commodities share a common source vertex, the unsplittable flow problem gets considerably easier. Results for the single source unsplittable flow problem have, for example, been obtained in [7,11,12,16].

<sup>1</sup> Unless stated otherwise, the balance condition is always assumed to be met for the UFP.

As mentioned above the  $k$ -SFP was introduced by Baier, Köhler, and Skutella [4]. They first consider a special form of the  $k$ -SFP in which the task is to find a feasible  $k$ -split routing that uses exactly  $k_i$  paths for commodity  $i \in \{1, \dots, K\}$  which all carry the same amount of flow. They call this problem the *uniform exactly- $k$ -SFP*. Since this problem still contains the UFP as a special case, it is also NP-hard. But note that any instance of it can be solved by solving a special instance of the UFP on the same graph. Thus, it holds that any  $\rho$ -approximation algorithm for the problem of minimizing the congestion for the UFP provides a  $\rho$ -approximation algorithm for the problem of minimizing the congestion for this special  $k$ -SFP. To approximate an instance of the general  $k$ -SFP Baier et al. use the fact that a uniform exactly- $k$ -split routing of minimal congestion for any instance of the  $k$ -SFP is a  $k$ -split routing which approximates the congestion of an optimal  $k$ -split routing for the same instance within a factor 2.

Bagchi, Chaudhary, Scheideler, and Kolman [3] introduced a problem which is quite similar to the  $k$ -SFP. They consider fault tolerant routings in networks. To ensure connection for each commodity for up to  $k - 1$  edge failures in the network, they require  $k \in \mathbb{N}$  edge disjoint flow paths per commodity. This problem is called the  $k$ -disjoint flow problem ( $k$ -DFP). Bagchi et al. [3] also consider the integral splittable flow problem (ISF). Bagchi [2] even extended the considerations of [3] to the  $k$ -SFP. The aim of Bagchi et al. is always to maximize the sum of satisfied demands subject to meeting all edge capacity constraints. In the  $k$ -DFP a demand  $d$  for any request is to be satisfied by a flow on  $k$  disjoint paths, each path carrying  $d/k$  units of flow. In the ISF integral demands need to be satisfied by an arbitrary number of paths, where the flow value of any path has to be integral. In contrast to [4] and the present paper, Bagchi does not admit different bounds on the numbers of paths for different commodities in the  $k$ -SFP. For all of the mentioned problems, Bagchi et al. introduce simple greedy algorithms in the style of greedy algorithms for the UFP given by Kolman and Scheideler [13]. With these algorithms they obtain approximation ratios of  $O(k^3 F \log(kF))$  for the  $k$ -DFP and  $O(k^2 F)$  for the  $k$ -SFP on the conditions, that they have unit capacities and the maximum demand is at most  $k$  times larger than the minimum edge capacity. Here  $F$  is again the flow number of the considered instance of the according problem. For the ISF Bagchi et al. obtain an approximation ratio of  $O(F)$  for any instance with uniform capacities in which the maximum demand is at most the capacity of the edges. The ISF has earlier been studied by Guruswami et al. [9] who obtained an approximation ratio of  $O(\sqrt{m\Delta} \log^2 m)$  for it, where  $\Delta$  is the maximum degree of a vertex in the considered graph.

*Contribution of this paper.* Randomized rounding has proved to be a very successful tool in the design of approximation algorithms for many NP-hard discrete optimization problems during the last decade. The general idea is as follows: Formulate the problem as a 0/1-integer linear program, solve the linear programming relaxation, and randomly turn the resulting fractional solution into an integral solution by interpreting the fractional values as probabilities. This idea was originally introduced in 1987 by Raghavan and Thompson [15] for the edge-disjoint paths problem which is a special case of the unsplittable flow prob-

lem. Although our general understanding of approximability has considerably improved since then, the  $O(\log m)$ -approximation for the minimum congestion version of the UFP is still best known. Even worse, it is neither known that no constant factor approximation exists (unless  $P=NP$ ) nor has any non-trivial lower bound on the performance of randomized rounding been achieved. In Section 3 we prove that the performance ratio of randomized rounding for the UFP is  $\Omega(\log m / \log \log m)$ .

In Section 2 we define the  $k$ -splittable flow problem with path capacities. For both variants of the problem discussed above, we prove that they have essentially the same approximability as the unsplittable flow problem. To be more precise, we show that any  $\rho$ -approximation algorithm for the UFP can be turned into a  $2\rho$ -approximation for either variant of our problem. The underlying idea is to decompose the solution of the problem into two parts. First, a packing of flow into containers is obtained. Then, each container is routed along a path through the network. The latter step can be formulated as an unsplittable flow problem. In Section 4 and Section 5, respectively, we present simple packing routines for both variants of the problem. We show that we do not lose more than a factor 2 with respect to congestion. Due to space limitations, we omit some details in this extended abstract.

## 2 Problem Definition and Notation

An instance of the  $k$ -SFP with path capacities consists of a digraph  $G = (V, E)$  with edge capacities  $c : E \rightarrow \mathbb{R}^+$  and a set  $\mathcal{T} \subseteq V \times V$  of  $K \in \mathbb{N}$  requests or commodities. The  $i$ th request is denoted by  $(s_i, t_i)$ . Together with request  $i$  we are given its non-negative demand  $d_i$  and the number of paths  $k_i$  (containers) it can be routed on. The flow value on the  $j$ th path of commodity  $i$  must be bounded by the given path capacity  $t_j^i \geq 0$ . We always assume that the path capacities of every commodity suffice to route the entire demand, that is,  $\sum_{j=1}^{k_i} t_j^i \geq d_i$ , for all  $i = 1, \dots, K$ .

A feasible solution to the  $k$ -SFP with path capacities consists of  $s_i$ - $t_i$ -paths  $P_1^i, \dots, P_{k_i}^i$  with corresponding nonnegative flow values  $f_1^i, \dots, f_{k_i}^i$  for each commodity  $i$ , such that the following requirements are met:

$$\sum_{j=1}^{k_i} f_j^i = d_i \quad \text{for all } i \in \{1, \dots, K\}, \quad (1)$$

that is, the whole demand of request  $i$  is routed. Moreover,

$$f_j^i \leq t_j^i \quad \text{for all } i \in \{1, \dots, K\} \text{ and } j \in \{1, \dots, k_i\}, \quad (2)$$

that is, the path capacities are obeyed. Finally,

$$\sum_{i=1}^K \sum_{\substack{j=1, \dots, k_i: \\ e \in P_j^i}} f_j^i \leq c(e) \quad \text{for all } e \in E, \quad (3)$$

or

$$\sum_{i=1}^K \sum_{\substack{j=1, \dots, k_i: \\ f_j^i > 0 \wedge e \in P_j^i}} t_j^i \leq c(e) \quad \text{for all } e \in E, \quad (4)$$

where (3) must hold if we consider the problem with *weight capacities*, and (4) must hold if we consider the problem with *size capacities*. That is, the according inequalities ensure that no edge capacity constraint is violated. We do not require that the paths  $P_1^i, \dots, P_{k_i}^i$  are distinct, for  $i \in \{1, \dots, K\}$ . Furthermore, we allow a path to have flow value 0. In other words, we may use less than  $k_i$  paths for commodity  $i$ .

In the following we use the notion *routing* for a pair  $(\mathcal{R}, (f_P)_{P \in \mathcal{R}})$ , where  $\mathcal{R}$  is a set of paths in the considered graph and  $f_P$  is the flow value on path  $P \in \mathcal{R}$ . We will use the notion *k-split routing* for a routing of  $\mathcal{T}$  that routes the whole demand of each commodity and uses at most  $k_i$  paths for commodity  $i$ . A *feasible k-split routing* shall be one which additionally satisfies all edge capacity constraints.

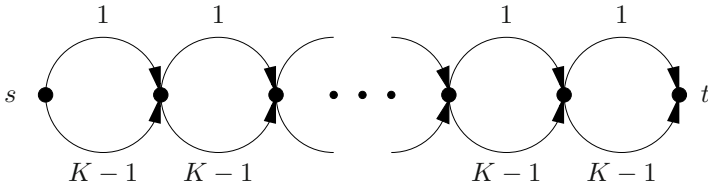
Note, that the k-SFP with path capacities is NP-hard, because it contains the unsplittable flow problem as a special case (set  $k_i = 1$  and  $t_1^i = d_i$ , for all  $i = 1, \dots, K$ ). Even the single commodity version is NP-hard since it is a generalization of the NP-hard single commodity k-SFP [4]. Therefore we are interested in approximate solutions to the problem. The *congestion* of a routing is the maximum factor by which the capacity of an edge in the graph is violated. Consider any k-split routing. For either variant of the k-SFP, we can simply define the congestion of this routing as the smallest value  $\alpha \geq 1$  for which multiplying the right hand side of (3) (or (4), respectively) makes this inequality true.

### 3 A Lower Bound for Randomized Rounding

In this section we consider the unsplittable flow problem, i.e., the special case  $k_i = 1$  and  $t_1^i = d_i$ , for all  $i = 1, \dots, K$ . We present a family of instances for which the congestion of an unsplit routing found by randomized rounding is always a factor  $\Omega(\log m / \log \log m)$  away from the congestion of an optimum solution. The performance of randomized rounding depends on the particular choice of an optimum fractional solution. We assume that an optimum fractional solution with a minimum number of paths is used.

**Theorem 1.** *The performance ratio of randomized rounding for the minimum congestion version of the unsplittable flow problem is  $\Omega(\log m / \log \log m)$ . The result even holds if we start with an optimum fractional solution with a minimum number of paths.*

*Proof.* Consider the instance of the UFP depicted in Figure 1. Obviously, there exists an unsplit routing with congestion 1 where the flow value on any edge is equal to its capacity. Of course, the latter property must also hold for any



**Fig. 1.** An instance of the unsplittable flow problem consisting of  $K$  commodities with unit demands, sharing a common source  $s$  and a common sink  $t$ . There are  $K^K$  consecutive pairs of parallel edges. The numbers at the edges indicate capacities.

optimum fractional solution. Consider the following fractional path decomposition: Each commodity  $i = 1, \dots, K$  is distributed among  $K$  paths  $P_1^i, \dots, P_K^i$  with flow values equal to  $1/K$ . The  $K^K$  upper edges of the graph are labeled by  $K$ -tuples  $(j_1, \dots, j_K) \in \{1, \dots, K\}^K$  such that every edge gets a different label. For  $i, j = 1, \dots, K$ , path  $P_j^i$  uses an upper edge labeled  $(j_1, \dots, j_K)$  if and only if  $j_i = j$ . The underlying intuition of this construction is that, for each subset of paths containing exactly one path of each commodity, there is an upper edge that is exactly used by the paths in the considered subset.

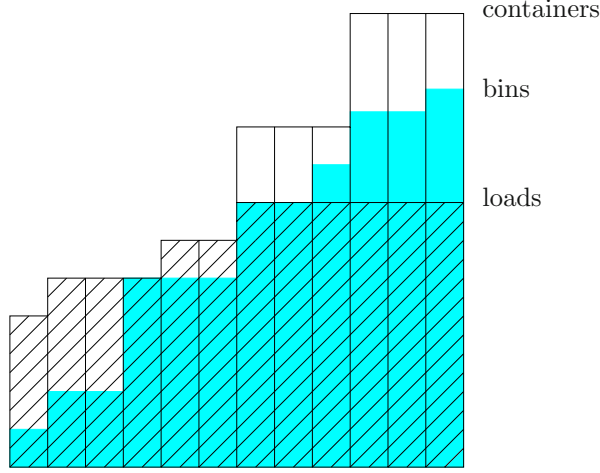
We argue that, for the described path decomposition, randomized rounding always yields an unsplit routing of congestion  $K$ : By construction of the fractional path decomposition, for every possible random choice, there exists an upper edge of unit capacity which is used by all  $K$  commodities. In particular, the flow on this edge exceeds its capacity by a factor  $K$ . Since  $m = 2K^K$ , we get  $K = \Omega(\log m / \log \log m)$ .

Unfortunately, the fractional path decomposition used above is somehow artificial. By perturbing arc capacities and demand values one can ensure that our path decomposition is minimal and still yields the same lower bound. Further details are omitted due to space limitations.  $\square$

## 4 The k-Splittable Flow Problem with Weight Capacities

In this section we consider the first variant of the k-SFP with path capacities, i.e., the case of *weight capacities*. We present a simple packing strategy which allows us to reduce the k-SFP with path capacities to the UFP while losing at most a factor 2 with respect to optimum congestion. We decompose the demand  $d_i$  of commodity  $i$  into  $k_i$  pieces  $load_1^i, \dots, load_{k_i}^i$  and then replace request  $i$  by  $k_i$  copies of  $i$ , whose demands are set to the values of the demand pieces. To make the procedure more vivid, one may always think of packing  $k_i$  containers for each commodity  $i$  which are then routed along certain paths through the network.

After defining the load for each container, we consider an optimum solution to the k-SFP and interpret its flow values as capacities of bins. This interpretation comes from the idea that we can assign our loads to these resulting bins for the purpose of sending each load (container) along the same path as the



**Fig. 2.** An example for the loading of the containers. The beams represent the containers, the striped part the loading defined in (5), and the gray filled part the loads of an optimal solution.

corresponding bin is taking in the optimum solution. We show that one can find an assignment of loads to bins such that the bins are overloaded by a factor of at most 2. Consequently, there exists a solution which uses our packing of containers and has a congestion of at most twice the minimum congestion.

For the purpose of assigning the loads to the bins with a small overload factor, it appears to be reasonable to break the demand of any commodity into pieces which are as small as possible. In other words, we want to load the containers such that the maximum load is minimized. One can visualize this by picturing that all containers of the same commodity are lined up in a row and merged with each other and get filled up with water; see Figure 2. To simplify notation, we assume without loss of generality that  $t_1^i \leq t_2^i \leq \dots \leq t_{k_i}^i$ , for all  $i = 1, \dots, K$ . Then, we can assume that the loads  $load_j^i$  of commodity  $i$  are indexed in nondecreasing order, too. A precise description of the loading strategy depicted in Figure 2 is as follows:

$$load_j^i = \min \left\{ t_j^i, \frac{1}{k_i - j + 1} \left( d_i - \sum_{j'=1}^{j-1} load_{j'}^i \right) \right\}. \quad (5)$$

For any  $i \in \{1, \dots, K\}$ , we can simply start with  $j = 1$  and then calculate the loads in increasing order. Note that two containers of same size get the same load. Thus, it even suffices to calculate the load for only one container per size.

Now consider any optimum solution  $F$  to the underlying problem of minimizing the congestion for the k-SFP with path capacities. Denote its flow values by  $bin_1^i, \dots, bin_{k_i}^i$ , for all  $i = 1, \dots, K$ . If we arrange the bins in nondecreasing order, for all  $i = 1, \dots, K$ , then we can prove the following lemma.



**Lemma 1.** *For all  $i \in \{1, \dots, K\}$  the following holds: If for some  $j \in \{1, \dots, k_i\}$  it holds that  $\text{load}_j^i > \text{bin}_j^i$ , then  $\text{load}_j^i \geq \text{bin}_{j'}^i$ , for all  $j' < j$ .*

*Proof.* Since  $F$  is an optimal and thus feasible solution, no bin is larger than its path capacity. Hence, a violation of the claim would contradict our loading strategy; see also Figure 2.  $\square$

Next we prove that we can pack  $k_i$  items of size  $\text{load}_1^i, \dots, \text{load}_{k_i}^i$  into  $k_i$  bins of size  $\text{bin}_1^i, \dots, \text{bin}_{k_i}^i$  without exceeding the bin sizes by more than a factor 2. This implies that we can simultaneously route all demands  $d_i$  in the instance of the UFP mentioned above, in spite of not allowing a congestion greater than twice the congestion of the optimum solution  $F$ . (Remember, that the instance of the UFP resulted from replacing each request in  $\mathcal{T}$  by exactly  $k_i$  copies and giving the copies the demands  $\text{load}_1^i, \dots, \text{load}_{k_i}^i$ .) The proof of the following lemma is omitted due to space limitations.

**Lemma 2.** *Let  $a_1 \leq a_2 \leq \dots \leq a_n$  and  $b_1 \leq b_2 \leq \dots \leq b_n$  be nonnegative real numbers with  $\sum_{i=1}^n a_i = \sum_{i=1}^n b_i$  and satisfying the following: If  $a_I > b_I$  for some  $I \in \{1, \dots, n\}$ , then  $a_i \geq b_i$  for all  $i = 1, \dots, I$ . Then there exists an assignment  $f : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$  such that  $\sum_{j \in f^{-1}(i)} a_j \leq 2b_i$ , for all  $i = 1, \dots, n$ .*

Note that for each  $i \in \{1, \dots, K\}$  the loads and the bins satisfy the requirements from Lemma 2. Thus, we can assign all loads to the bins without exceeding the bin capacities by a factor greater than 2. Now consider the following instance of the UFP: Take the underlying k-SFP with path capacities. For all  $i \in \{1, \dots, K\}$ , replace request  $i$  by  $k_i$  copies of  $i$  with demands equal to  $\text{load}_1^i, \dots, \text{load}_{k_i}^i$ . Using the results from above, we obtain the following:

**Theorem 2.** *For the problem of minimizing congestion, a  $\rho$ -approximate solution to the instance of the UFP described above yields a  $2\rho$ -approximate solution to the underlying instance of the k-SFP with path capacities (weight capacities).*

We conclude this section with some remarks on the running time of the resulting  $2\rho$ -approximation algorithms for the k-SFP with path capacities. Notice that the size of the instance of the UFP described above is not necessarily polynomial in the input size. The number of copies  $k_i$  of request  $i$  can be exponential, if we are only given the different sizes of containers and the number of containers for each size in the input of the k-SFP with path capacities. As already mentioned above, it will not be a problem to compute the loads in polynomial time. However, in order to achieve polynomial running time when solving the resulting instance of the UFP, we must use a polynomial time algorithm for the UFP which is able to handle a compact input format where, for each request, the number of copies of the request is given. In fact, most approximation algorithms for the UFP fulfill this requirement or can easily be modified to fulfill it. As an example, we mention the randomized rounding procedure: In a fractional

solution, all copies of a request can be treated as one commodity for which a path decomposition using at most  $m$  paths exists. Randomized rounding then assigns to each of these paths a certain number of copies of the corresponding request. This procedure can easily be implemented to run in polynomial time.

## 5 The k-Splittable Flow Problem with Size Capacities

In this section we consider the second variant of the k-SFP with path capacities, i.e., the case of *size capacities*. If all path capacities are equal to some uniform value  $T_i$ , for all commodities  $i$ , it is easy to observe that there always exists an optimum solution using the minimum number of paths  $\lceil d_i/T_i \rceil$  for commodity  $i$ . Therefore, this problem can be formulated as a ‘uniform exactly-k-SFP’ (see [4] for details). In particular, all results for the latter problem obtained in [4] also hold for the k-SFP with uniform path capacities.

We turn to the general problem. As in the last section, we assume that  $t_1^i \leq \dots \leq t_{k_i}^i$ , for all  $i = 1, \dots, K$ . Our general approach is identical to the one presented in the last section. Again, we first give a simple strategy for the packing of containers. This reduces the problem to an UFP. We prove that at most a factor 2 in the performance is lost due to this packing by comparing it to the packing of an optimum solution (‘bins’).

In contrast to the last section, it seems to be reasonable to load each container that is used up to its size. Notice that only the size of a container (and not its actual load) matters for the routing. Among all containers, we try to use the smallest ones in order to be able to obtain reasonable assignments to bins later. The load  $load_j^i$  of the  $j$ th container of commodity  $i$  is determined as follows: Consider the containers of each commodity in order of decreasing size. If the whole (remaining) demand of the commodity fits into the remaining containers with smaller index, then discard the container. Otherwise, the container is filled up to its size (unless the remaining demand is smaller than the size) and the remaining demand is decreased by this value.

The above strategy finds the loads in polynomial time if the path capacity of every path is explicitly given in the input. If the encoding of the input is more compact and, for each commodity  $i$ , only specifies the different path capacities together with the number of paths for each capacity, the algorithm can easily be adapted to still run in polynomial time.

In the following we consider a fixed optimum solution to the k-SFP with path capacities. For all  $i \in \{1, \dots, K\}$ , let  $\mathcal{O}_i \subseteq \{1, \dots, k_i\}$  such that  $j \in \mathcal{O}_i$  if and only if the  $j$ th container is used by the optimum solution. Similarly, let  $\mathcal{B}_i \subseteq \{1, \dots, k_i\}$  such that  $j \in \mathcal{B}_i$  if and only if our packing uses the  $j$ th container, i.e.,  $load_j^i > 0$ . The following lemma says that the containers used by our loading can be packed into the containers used by the optimum solution such that the size of any container of the optimum solution is exceeded by at most a factor 2.

**Lemma 3.** *For all  $i = 1, \dots, K$ , there exists an assignment  $f : \mathcal{B}_i \rightarrow \mathcal{O}_i$  such that*

$$\sum_{j' \in f^{-1}(j)} t_{j'}^i \leq 2t_j^i \quad \text{for all } j \in \mathcal{O}_i. \quad (6)$$

*Proof.* In the following we keep  $i$  fixed; to simplify notation, we omit all indices and superscripts  $i$ . By construction of our packing strategy,

$$\sum_{j \in \mathcal{B}: j > j_0} t_j < \sum_{j \in \mathcal{O}: j \geq j_0} t_j \quad \text{for all } j_0 \in \mathcal{B}. \quad (7)$$

In particular, it follows from (7) that the maximum index in  $\mathcal{O}$  is at least as large as the maximum index in  $\mathcal{B}$  and therefore the largest container in the optimum solution is at least as large as the largest container in our packing. While  $\mathcal{B} \neq \emptyset$ , we construct the assignment  $f$  iteratively as follows:

$$j_{\max} := \max\{j \mid j \in \mathcal{O}\} \quad \text{and} \quad \bar{j} := \min\{j \in \mathcal{B} \mid \sum_{j' \in \mathcal{B}: j' > j} t_{j'} \leq t_{j_{\max}}\}.$$

Set  $f(j) := j_{\max}$  for all  $j \in \mathcal{B}$  with  $j \geq \bar{j}$ ; set  $\mathcal{O} := \mathcal{O} \setminus \{j_{\max}\}$  and  $\mathcal{B} := \mathcal{B} \setminus \{j \mid j \geq \bar{j}\}$ . Notice that property (6) holds for  $j_{\max}$  since, by (7), the container size  $t_{j_{\max}}$  is an upper bound on  $t_j$ , for all  $j \in \mathcal{B}$ . Moreover, (7) is still fulfilled for the reduced sets  $\mathcal{B}$  and  $\mathcal{O}$ . This concludes the proof.  $\square$

For all  $i = 1, \dots, K$  and  $j = 1, \dots, k_i$ , set  $\tilde{d}_j^i := t_j^i$  if  $j \in \mathcal{B}_i$ , and  $\tilde{d}_j^i := 0$ , otherwise. We define an instance of the UFP by replacing each request  $i$  by  $k_i$  copies with demands  $\tilde{d}_1^i, \dots, \tilde{d}_{k_i}^i$ . According to Lemma 3, there exists a solution to this instance of the UFP whose congestion is at most twice the minimum congestion for the underlying instance of the k-SFP with path capacities.

**Theorem 3.** *For the problem of minimizing congestion, a  $\rho$ -approximate solution to the instance of the UFP described above yields a  $2\rho$ -approximate solution to the underlying instance of the k-SFP with path capacities (size capacities).*

With respect to the running time of the resulting  $2\rho$ -approximation algorithms for the k-SFP with path capacities, the same remarks hold that we made in the last section after Theorem 2.

We conclude with the discussion of a special case of the k-SFP with path capacities where, for each request  $i$ , each of its path capacities is a multiple of all of its smaller path capacities. This property holds, for example, if all path capacities are powers of 2. In this case, we can use the same loading strategy as above in order to obtain a  $\rho$ -approximate solution to the underlying problem from any  $\rho$ -approximation algorithm for the UFP. Notice that there is an optimal solution to the underlying problem which uses the containers given by  $\mathcal{B}_i$ . The argument is similar to that in the proof of Lemma 3.

**Acknowledgements.** Part of this work was done while the authors were visiting Escuela Politécnica Nacional in Quito, Ecuador. We would like to thank Walter Polo Vaca Arellano and Luis Miguel Torres Carvajal for their hospitality.

## References

1. Y. Azar and O. Regev. Strongly polynomial algorithms for the unsplittable flow problem. In *Proceedings of the 8th Conference on Integer Programming and Combinatorial Optimization*, pages 15–29, 2001.
2. A. Bagchi. *Efficient Strategies for Topics in Internet Algorithmics*. PhD thesis, The Johns Hopkins University, October 2002.
3. A. Bagchi, A. Chaudary, C. Scheideler, and P. Kolman. Algorithms for fault-tolerant routing in circuit switched networks. In *Fourteenth ACM Symposium on Parallel Algorithms and Architectures*, 2002.
4. G. Baier, E. Köhler, and M. Skutella. On the k-splittable flow problem. *Algorithmica*. To appear. An extended abstract appeared in Rolf H. Möhring and Rajeev Raman (eds.): *Algorithms - ESA '02*, Lecture Notes in Computer Science 2461, Springer: Berlin, 2002, 101–113, Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02).
5. A. Baveja and A. Srinivasan. Approximation algorithms for disjoint paths and related routing and packing problems. *Mathematics of Operations Research*, 25, pages 255–280, 2000.
6. C. Chekuri and S. Khanna. Edge disjoint paths revisited. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.
7. Y. Dinitz, N. Garg, and M. X. Goemans. On the single source unsplittable flow problem. *Combinatorica*, 19:17–41, 1999.
8. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.
9. V. Guruswami, S. Khanna, R. Rajaraman, B. Shepherd, and M. Yannakakis. Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 19–28, 1999.
10. J. Kleinberg and R. Rubinfeld. Short paths in expander graphs. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 86–95, 1996.
11. J. M. Kleinberg. *Approximation Algorithms for Disjoint Path Problems*. PhD thesis, Massachusetts Institute of Technology, May 1996.
12. S. G. Kolliopoulos and C. Stein. Approximation algorithms for single-source unsplittable flow. *SIAM Journal on Computing*, 31:919–946, 2002.
13. P. Kolman and C. Scheideler. Improved bounds for the unsplittable flow problem. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 184–193, 2002.
14. P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Sciences*, 37, pages 130–143, 1988.
15. P. Raghavan and C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7, pages 365–374, 1987.
16. M. Skutella. Approximating the single source unsplittable min-cost flow problem. *Mathematical Programming*, 91:493–514, 2002.

# Maximum Matchings in Planar Graphs via Gaussian Elimination<sup>\*</sup>

(Extended Abstract)

Marcin Mucha and Piotr Sankowski

Institute of Informatics, Warsaw University,  
Banacha 2, 02-097, Warsaw, Poland {mucha,sank}@mimuw.edu.pl

**Abstract.** We present a randomized algorithm for finding maximum matchings in planar graphs in time  $O(n^{\omega/2})$ , where  $\omega$  is the exponent of the best known matrix multiplication algorithm. Since  $\omega < 2.38$ , this algorithm breaks through the  $O(n^{1.5})$  barrier for the matching problem. This is the first result of this kind for general planar graphs. Our algorithm is based on the Gaussian elimination approach to maximum matchings introduced in [1].

## 1 Introduction

A *matching* of an undirected graph  $G = (V, E)$  is a subset  $M \subseteq E$ , such that no two edges in  $M$  are incident. Let  $n = |V|$ ,  $m = |E|$ . A *perfect matching* is a matching of cardinality  $n/2$ . The problems of finding a *Maximum Matching* (i.e. a matching of maximum size) and, as a special case, finding a *Perfect Matching* if one exists, are two of the most fundamental algorithmic problems.

Solving these problems in time polynomial in  $n$  remained an elusive goal for a long time until Edmonds [2] gave the first algorithm. Several other algorithms have been found since then, the fastest of them being the algorithm of Micali and Vazirani [3], Blum [4] and Gabow and Tarjan [5]. The first of these algorithms is in fact a modification of the Edmonds algorithm, the other two use different techniques, but all of them run in time  $O(m\sqrt{n})$ , which gives  $O(n^{2.5})$  for dense graphs.

The matching problems seem to be inherently easier for planar graphs. For a start, these graphs have  $O(n)$  edges, so  $O(m\sqrt{n}) = O(n^{1.5})$ . But there is more to it. Using the duality-based reduction of maximum flow with multiple sources and sinks to single source shortest paths problem [6], Klein et al. [7] were able to give an algorithm finding perfect matchings in bipartite planar graphs in time  $O(n^{\frac{4}{3}} \log n)$ . This reduction, however, does not carry over to the case of general planar graphs.

We have recently shown [1], that extending the randomized technique of Lovász [8] leads to an  $O(n^{\omega})$  algorithm for finding maximum matching in general graphs. In this paper we use similar techniques, together with separator based

---

<sup>\*</sup> Research supported by KBN grant 4T11C04425.

decomposition of planar graphs and the fast nested dissection algorithm, to show that maximum matchings in planar graphs can be found in time  $O(n^{\omega/2})$ .

*Remark 1.* In case  $\omega = 2$  additional logarithmic factor appears, so in the remainder of this paper we assume for simplicity, that  $\omega > 2$ .

There is one point to notice here. The  $O(n^\omega)$  algorithm for general graphs presented in [1] is faster than the standard maximum matching algorithms only if the Coppersmith-Winograd matrix multiplication is used (see [9]). On the other hand, for our  $O(n^{\omega/2})$  algorithm to be faster than the standard algorithms applied to planar graphs, it is enough to use any  $o(n^3)$  matrix multiplication algorithm, e.g. the classic algorithm of Strassen [10]. This suggests that our results not only constitute a theoretical breakthrough, but might also give a new practical approach to solving the maximum matching problem in planar graphs.

The same techniques can be used to generate perfect matchings in planar graphs uniformly at random using  $O(n^{\omega/2})$  arithmetic operations. This extends the result of Wilson [11]. Details will be given in the full version of this paper.

The rest of the paper is organized as follows. In the next section we recall some well known results concerning the algebraic approach to the maximum matching problem and the key ideas from [1]. In section 3 we recall the separator theorem for planar graphs and the fast nested dissection algorithm and show how these can be used to test planar graphs for perfect matchings in time  $O(n^{\omega/2})$ . In Section 4, we present an algorithm for finding perfect matchings in planar graphs in time  $O(n^{\omega/2})$ , and in Section 5 we show how to extend it to an algorithm finding maximum matchings. In all these algorithms we use rational arithmetic and so their time complexity is in fact larger than  $O(n^{\omega/2})$ . This issue is addressed in Section 6, where we show, that all the computations can be performed over a finite field  $\mathbb{Z}_p$ , for a random prime  $p = \Theta(n^4)$ .

## 2 Preliminaries

### 2.1 Matchings, Adjacency Matrices, and Their Inverses

Let  $G = (V, E)$  be a graph and let  $n = |V|$  and  $V = \{v_1, \dots, v_n\}$ . A *skew symmetric adjacency matrix* of  $G$  is a  $n \times n$  matrix  $\tilde{A}(G)$  such that

$$\tilde{A}(G)_{i,j} = \begin{cases} x_{i,j} & \text{if } (v_i, v_j) \in E \text{ and } i < j \\ -x_{i,j} & \text{if } (v_i, v_j) \in E \text{ and } i > j \\ 0 & \text{otherwise} \end{cases},$$

where the  $x_{i,j}$  are unique variables corresponding to the edges of  $G$ .

Tutte [12] observed the following

**Theorem 2.** *The symbolic determinant  $\det \tilde{A}(G)$  is non-zero iff  $G$  has a perfect matching.*

Lovász[8] generalized this to

**Theorem 3.** *The rank of the skew symmetric adjacency matrix  $\tilde{A}(G)$  is equal to twice the size of maximum matching of  $G$ .*

Choose a number  $R = n^{O(1)}$  (more on the choice of  $R$  in Section 6) and substitute each variable in  $\tilde{A}(G)$  with a random number taken from the set  $\{1, \dots, R\}$ . Let us call the resulting matrix the *random adjacency matrix* of  $G$  and denote  $A(G)$ . Lovász showed that

**Theorem 4.** *The rank of  $A(G)$  is at most twice the size of maximum matching in  $G$ . The equality holds with probability at least  $1 - (n/R)$ .*

This gives a randomized algorithm for deciding whether a given graph  $G$  has a perfect matching: Compute the determinant of  $A(G)$ . With high probability, this determinant is non-zero iff  $G$  has a perfect matching. More precisely, if the determinant is non-zero then  $G$  has a perfect matching, but with small probability the determinant can be zero even if  $G$  has a perfect matching. This algorithm can be implemented to run in time  $O(n^\omega)$  using fast matrix multiplication (where  $\omega$  is the matrix multiplication exponent, currently  $\omega < 2.38$ , see Coppersmith and Winograd [9]).

Let  $G$  be a graph having a perfect matching and let  $A = A(G)$  be its random adjacency matrix. With high probability  $\det A \neq 0$ , and so  $A$  is invertible. Rabin and Vazirani [13] showed that

**Theorem 5.** *With high probability,  $(A^{-1})_{j,i} \neq 0$  iff the graph  $G - \{v_i, v_j\}$  has a perfect matching.*

In particular, if  $(v_i, v_j)$  is an edge in  $G$ , then with high probability  $(A^{-1})_{j,i} \neq 0$  iff  $(v_i, v_j)$  is *allowed*, i.e. it is contained in some perfect matching. This follows from the formula  $(A^{-1})_{i,j} = \text{adj}(A)_{i,j} / \det A$ , where  $\text{adj}(A)_{i,j}$  — the so called adjoint of  $A$  — is the determinant of  $A$  with the  $j$ -th row and  $i$ -th column removed, multiplied by  $(-1)^{i+j}$ .

## 2.2 Randomization

Recall the classic lemma due to Zippel [14] and Schwartz [15]

**Lemma 6.** *If  $p(x_1, \dots, x_m)$  is a non-zero polynomial of degree  $d$  with coefficients in a field and  $S$  is a subset of the field, then the probability that  $p$  evaluates to 0 on a random element  $(s_1, s_2, \dots, s_m) \in S^m$  is at most  $d/|S|$ .*

Notice that the determinant of the symbolic adjacency matrices defined in the previous subsection is in fact a polynomial of degree  $n$ . Thus, if we substitute random numbers from the set  $\{1, \dots, R\}$  for the variables appearing in these matrices, then the probability of getting a false zero is smaller than  $n/R$ . This is exactly Theorem 4.

The Zippel-Schwartz Lemma can be used to argue that with high-probability no false zeros appear in our algorithms. We will provide the details in the full

version of this paper. Throughout the rest of this paper, we omit the “with high probability” phrase in most statements.

Computations performed by our algorithms rely on the underlying field having characteristic zero. That is why we present them over the field of rationals. For simplicity of presentation, throughout the paper we assume that we can perform rational arithmetic operations in constant time, which allows us to avoid differentiating between the time complexity and the number of arithmetic operations. In Section 6, we use the Zippel-Schwartz Lemma to show that our algorithms can be performed over  $\mathcal{Z}_p$  for large enough (polynomial) random prime  $p$ .

### 2.3 Perfect Matchings via Gaussian Elimination

We now recall a technique, recently developed by the authors, of finding perfect matchings using Gaussian elimination. This technique can be used to find an inclusion-wise maximal allowed submatching of any matching in time  $O(n^\omega)$ , which is a key element of our matching algorithm for planar graphs. A more detailed exposition of the Gaussian elimination technique and faster algorithm for matchings in bipartite and general graphs can be found in [1].

Consider a random adjacency matrix  $A = A(G)$  of a graph  $G = (V, E)$ , where  $|V| = n$ ,  $V = \{v_1, v_2, \dots, v_n\}$ . If  $(v_i, v_j) \in E$  and  $(A^{-1})_{i,j} \neq 0$ , then  $(v_i, v_j)$  is an allowed edge. We may thus choose this edge as a matching edge and try to find a perfect matching in  $G' = G - \{v_1, v_2\}$ . The problem with this approach is that edges that were allowed in  $G$  might not be allowed in  $G'$ . Computing the matrix  $A(G')^{-1}$  from scratch is out of the question as this would lead to a  $O(n^{\omega+1})$  algorithm for perfect matchings. There is however another way suggested by the following well known property of Schur complement

**Theorem 7 (Elimination theorem).** *Let*

$$A = \left( \begin{array}{c|c} a_{1,1} & v^T \\ \hline u & B \end{array} \right) \quad A^{-1} = \left( \begin{array}{c|c} \hat{a}_{1,1} & \hat{v}^T \\ \hline \hat{u} & \hat{B} \end{array} \right),$$

where  $\hat{a}_{1,1} \neq 0$ . Then  $B^{-1} = \hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1}$ .

The modification of  $\hat{B}$  described in this theorem is in fact a single step of the well-known Gaussian elimination procedure. In this case, we are eliminating the first variable (column) using the first equation (row). Similarly, we can eliminate from  $A^{-1}$  any other variable (column)  $j$  using any equation (row)  $i$ , such that  $(A^{-1})_{i,j} \neq 0$ .

In [1] we show, that among consequences of Theorem 7 is a very simple  $O(n^3)$  algorithm for finding perfect matchings in general graphs (this is an easy corollary) as well as  $O(n^\omega)$  algorithms for finding perfect matchings in bipartite and general graphs. The last of these requires some additional structural techniques.



## 2.4 Matching Verification

We now describe another consequence of Theorem 7, one that is crucial for our approach to finding maximum matchings in planar graphs. In [1] we have shown that

**Theorem 8.** *Naive iterative Gaussian elimination without row or column pivoting can be implemented in time  $O(n^\omega)$  using a lazy updating scheme.*

*Remark 9.* The algorithm in Theorem 8 is very similar to the classic Hopcroft-Bunch algorithm [16]. Being iterative, our algorithm is better suited for proving Theorem 10, where we need to skip row-column pairs if the corresponding diagonal element is zero.

This algorithm has a very interesting application

**Theorem 10.** *Let  $G$  be a graph having a perfect matching. For any matching  $M$  of  $G$ , an inclusion-wise maximal allowed (i.e. extendable to a perfect matching) submatching  $M'$  of  $M$  can be found in time  $O(n^\omega)$ .*

*Proof.* Let  $M = \{(v_1, v_2), (v_3, v_4), \dots, (v_{k-1}, v_k)\}$  and let  $v_{k+1}, v_{k+2}, \dots, v_n$  be the unmatched vertices. We compute the inverse  $A(G)^{-1}$  of the random adjacency matrix of  $G$  and permute its rows and columns so that the row order is  $v_1, v_2, v_3, v_4, \dots, v_n$  and the column order is  $v_2, v_1, v_4, v_3, \dots, v_n, v_{n-1}$ . Now, perform Gaussian elimination of the first  $k$  rows and  $k$  columns using the algorithm of Theorem 8, but if the eliminated element is zero just skip to the next iteration. The eliminated rows and columns correspond to a maximal submatching  $M'$  of  $M$ .  $\square$

## 3 Testing Planar Graphs for Perfect Matching

In this section we show how planar graphs can be tested for perfect matching in time  $O(n^{\omega/2})$ . We use the nested dissection algorithm that performs Gaussian elimination in  $O(n^{\omega/2})$  time for a special class of matrices. The results presented in the next subsection are due to Lipton and Tarjan [17], and Lipton, Rose and Tarjan [18]. We follow the presentation in [19] as it is best suited for our purposes.

### 3.1 Sparse LU Factorization via Nested Dissection

We say that a graph  $G = (V, E)$  has a  $s(n)$ -separator family (with respect to some constant  $n_0$ ) if either  $|V| \leq n_0$ , or by deleting a set  $S$  of vertices such that  $|S| \leq s(|V|)$ , we may partition  $G$  into two disconnected subgraphs with the vertex sets  $V_1$  and  $V_2$ , such that  $|V_i| \leq 2/3|V|$ ,  $i = 1, 2$ , and furthermore each of the two subgraphs of  $G$  defined by the vertex sets  $S \cup V_i$ ,  $i = 1, 2$  also has an  $s(n)$ -separator family. The set  $S$  in this definition is called an  $s(n)$ -separator in  $G$  (we also use the name *small separators* for  $O(\sqrt{n})$ -separators) and the partition

resulting from recursive application of this definition is the  $s(n)$ -separator tree. Partition of a subgraph of  $G$  defines its children in the tree.

The following theorem of Lipton, Rose and Tarjan [17] gives an important example of graphs having  $O(\sqrt{n})$ -separator families

**Theorem 11 (Separator theorem).** *Planar graphs have  $O(\sqrt{n})$ -separator families. Moreover, an  $\sqrt{n}$ -separator tree for a planar graph can be found in time  $O(n \log n)$ .*

Let  $A$  be an  $n \times n$  symmetric positive definite matrix. The graph  $G(A)$  corresponding to  $A$  is defined as follows:  $G(A) = (V, E)$ ,  $V = \{1, 2, \dots, n\}$ ,  $E = \{\{i, j\} | i \neq j \text{ and } A_{i,j} \neq 0\}$ . The existence of  $O(\sqrt{n})$ -separator family for  $G(A)$  makes faster Gaussian elimination possible as the following theorem of Lipton and Tarjan shows

**Theorem 12 (Nested dissection).** *Let  $A$  be a symmetric positive definite matrix and let  $G(A)$  have a  $O(\sqrt{n})$ -separator family. Given a  $O(\sqrt{n})$  separator tree for  $G(A)$ , Gaussian elimination on  $A$  can be performed in time  $O(n^{\omega/2})$  using the so-called nested dissection. The resulting LU factorization of  $A$  is given by matrices  $L$  and  $D$ ,  $A = LDL^T$ , where matrix  $L$  is unit lower-diagonal and has  $O(n \log n)$  non-zero entries and matrix  $D$  is diagonal.*

*Remark 13.* The assumption of  $A$  being symmetric positive definite is needed to assure that no diagonal zeros will appear. If we can guarantee this in some other way, then the assumption can be omitted.

We are not going to present the details of this algorithm. The basic idea is to permute rows and columns of  $A$  using the  $\sqrt{n}$ -separator tree. Vertices of the top-level separator  $S$  correspond to the last  $|S|$  rows and last  $|S|$  columns, etc.. When Gaussian elimination is performed in this order, matrix remains sparse throughout the elimination.

### 3.2 Vertex Splitting

The nested dissection algorithm cannot be applied to the random adjacency matrix  $A = A(G)$  of a planar graph. This matrix is neither symmetric nor positive definite. Fortunately, factorization of  $A$  is not necessary. For our purposes it is enough to factorize the matrix  $B = AA^T$ . Notice that this matrix is symmetric and if  $A$  is non-singular (i.e.  $G$  has a perfect matching), then  $B$  is positive definite.

We now need to show that  $G(B)$  and all its subgraphs have small separators, which needs not be true in general, but it is true if  $G$  is a bounded degree graph. It can be shown, using the standard technique of “vertex splitting” (see for example [11]) that it is sufficient to consider such graphs.

**Theorem 14.** *The problem of finding perfect (maximum) matchings in planar graphs is reducible in  $O(n)$  time to the problem of finding perfect (maximum) matchings in planar graphs with maximum vertex degree 3. This reduction adds  $O(n)$  new vertices.*

In order to use nested dissection on matrix  $B$  we have to show that the graph  $G(B)$  has  $O(\sqrt{n})$ -separator family. Let  $S$  be a small separator in  $G(A)$ , and consider the set  $T$  containing all vertices of  $S$  and all their neighbours. We call  $T$  a *thick separator corresponding to  $S$* . Notice that  $B_{i,j}$  can be non-zero only if there exists a path of length 2 between  $v_i$  and  $v_j$ . Thus  $T$  is a separator in  $G(B)$ .  $T$  is also a small separator, because  $G$  has bounded degree and so  $|T| \leq 4|S| = O(\sqrt{n})$ . In the same manner small separators can be found in any subgraph of  $G(B)$ , so LU factorization of  $B$  can be found using the nested dissection algorithm in time  $O(n^{\omega/2})$ .

### 3.3 The Testing Algorithm

The general idea of the testing algorithm is presented in Fig. 1. If the nested dissection algorithm finds an LU factorization of  $B$ , then  $B$  is non-singular, and so  $A$  is non-singular, thus  $G$  has a perfect matching. If, however, the nested dissection fails i.e. there appears zero on the diagonal during the elimination, then  $B$  is not positive definite, and so  $A$  is singular.

PLANAR-TEST-PERFECT-MATCHING( $G$ ):

1. reduce the degrees of vertices in  $G$ ;
2. compute  $B = AA^T$ ;
3. run nested dissection on  $B$ ;
4.  $G$  has a perfect matching iff the algorithm succeeds i.e. finds an LU factorization;

**Fig. 1.** An algorithm for testing if a planar graph has a perfect matching.

## 4 Finding Perfect Matchings in Planar Graphs

In this section we present an algorithm for finding perfect matchings in planar graphs. In Section 5 we show that the more general problem of finding a maximum matching reduces to the problem of finding a perfect matching.

### 4.1 The General Idea

For any matrix  $X$ , let  $X_{R,C}$  denote a submatrix of  $X$  corresponding to rows  $R$  and columns  $C$ .

The general idea of the matching algorithm is presented in Fig. 2.

To find a perfect matching in a planar graph, we find a small separator, match its vertices in an allowed way (i.e. one that can be extended to the set of all vertices), and then solve the problem for each of the connected components created by removing the endpoints of this matching. In the remainder of this section, we show that we can perform steps 3. and 4. in time  $O(n^{\omega/2})$ . This gives the complexity bound of  $O(n^{\omega/2})$  for the whole algorithm as well.

PLANAR-PERFECT-MATCHING( $G$ ):

1. run PLANAR-TEST-PERFECT-MATCHING( $G$ );
2. let  $S$  be a small separator in  $G$  and let  $T$  be the corresponding thick separator;
3. find  $(A(G)^{-1})_{T,T}$ ;
4. using the FIND-ALLOWED-SEPARATOR-MATCHING procedure, find an allowed matching  $M$  incident on all vertices of  $S$ ;
5. find perfect matchings in connected components of  $G - V(M)$ ;

**Fig. 2.** An algorithm for finding perfect matchings in planar graphs.

## 4.2 Computing the Important Part of $A(G)^{-1}$

We could easily find  $(A(G)^{-1})_{T,T}$  if we had an LU factorization of  $A = A(G)$ . Unfortunately,  $A$  is not symmetric positive definite, so we cannot use the fast nested dissection algorithm to factorize  $A$ . In the testing phase we find  $n \times n$  matrices  $L$  and  $D$  such that  $AA^T = LDL^T$ . We now show how  $L$  and  $D$  can be used to compute  $(A^{-1})_{T,T}$  in time  $O(n^{\omega/2})$ . Let us represent  $A$ ,  $L$  and  $D$  as block matrices

$$A = \left( \begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array} \right), \quad D = \left( \begin{array}{c|c} D_{1,1} & 0 \\ \hline 0 & D_{2,2} \end{array} \right),$$

$$L = \left( \begin{array}{c|c} L_{1,1} & 0 \\ \hline L_{2,1} & L_{2,2} \end{array} \right), \quad L^{-1} = \left( \begin{array}{c|c} L_{1,1}^{-1} & 0 \\ \hline -L_{2,2}^{-1}L_{2,1}L_{1,1}^{-1} & L_{2,2}^{-1} \end{array} \right),$$

where lower right blocks in all matrices correspond to the vertices of the thick separator  $T$ , for example  $A_{T,T} = A_{2,2}$ . Since  $AA^T = LDL^T$ , we have

$$(A^T)^{-1} = (L^T)^{-1}D^{-1}L^{-1}A,$$

where the interesting part of  $(A^T)^{-1}$  is

$$\begin{aligned} (A^T)_{T,T}^{-1} &= ((L^T)^{-1})_{T,V}D^{-1}L^{-1}A_{V,T} = \\ &= (L_{2,2}^T)^{-1}D_{2,2}^{-1}(L^{-1})_{T,V}A_{V,T} = \\ &= (L_{2,2}^T)^{-1}D_{2,2}^{-1}L_{2,2}^{-1}A_{2,2} + (L_{2,2}^T)^{-1}D_{2,2}^{-1}(L^{-1})_{2,1}A_{1,2}. \end{aligned}$$

The first component can be easily computed in time  $O(n^{\omega/2})$  using fast matrix multiplication. The second component can be written as

$$(L_{2,2}^T)^{-1}D_{2,2}^{-1}(L^{-1})_{2,1}A_{1,2} = -(L_{2,2}^T)^{-1}D_{2,2}^{-1}L_{2,2}^{-1}L_{2,1}L_{1,1}^{-1}A_{1,2}$$

and the only hard part here is to compute  $X = -L_{2,1}L_{1,1}^{-1}A_{1,2}$ . Consider the matrix

$$B = \left( \begin{array}{c|c} L_{1,1} & A_{1,2} \\ \hline L_{2,1} & 0 \end{array} \right).$$

When Gaussian elimination is performed on the non-separator columns and vertices of  $B$ , the lower right submatrix becomes  $X$ . This is a well known property of the Schur complement. The elimination can be performed with use of the nested dissection algorithm in time  $O(n^{\omega/2})$ . The idea here is that the separator tree for  $AA^T$  is a valid separator tree for  $L$ , thus also for  $B$ . The new non-zero entries of  $L$  introduced by Gaussian elimination (so called *fill-in*), correspond to the edges that can only go upwards in the separator tree, from child to one of its ancestors (see [20]). Notice, that since  $L_{1,1}$  is lower-diagonal, there are no problems with diagonal zeros, even though  $B$  is not symmetric positive definite.

### 4.3 Matching the Separator Vertices

We now show how the separator vertices can be matched using the matching verification algorithm. Consider the procedure presented in Fig. 3.

#### FIND-ALLOWED-SEPARATOR-MATCHING:

1. let  $M = \emptyset$ ;
2. let  $G_T = (T, E(T) - E(T - S))$ ;
3. let  $M_G$  be any inclusion-wise maximal matching in  $G_T$  using only allowed edges;
4. run the verification algorithm of Theorem 10 on  $(A^{-1})_{T,T}$  to find a maximal allowed submatching  $M'_G$  of  $M_G$ ;
5. add  $M'_G$  to  $M$ ;
6. remove the vertices matched by  $M'_G$  from  $G_T$ ;
7. mark edges in  $M_G - M'_G$  as not allowed;
8. if  $M$  does not match all vertices of  $S$  go to step 3.;

**Fig. 3.** A Procedure for finding allowed submatching of the separator.

The maximal allowed submatching  $M'_G$  of  $M_G$  can be found in time  $O(n^{\omega/2})$  using the verification algorithm. This algorithm requires the inverse  $A(G)^{-1}$  of the random adjacency matrix of  $G$ , but it never uses any values from outside the submatrix  $(A(G)^{-1})_{T,T}$  corresponding to the vertices of  $T$ , so we only have to compute this submatrix. Let  $\hat{A}$  be the result of running the verification algorithm on the matrix  $(A^{-1})_{T,T}$ . Notice that due to Theorem 7,  $\hat{A}_{T',T'} = (A^{-1})_{T',T'}$ , where  $T'$  is obtained from  $T$  by removing the vertices matched by  $M'_G$ . Thus the inverse does not need to be computed from scratch in each iteration of the loop.

Now, consider the allowed matching  $M$  covering  $S$ , found by the above algorithm. Notice that any edge  $e$  of  $M$  is either incident on at last one edge of the inclusion-wise maximal matching  $M_G$  or is contained in  $M_G$ , because of the maximality of  $M_G$ . If  $e$  is in  $M_G$ , it is chosen in step 4, otherwise one of the edges incident to  $e$  is marked as not allowed. Every edge  $e \in M$  has at most 4 incident edges, so the loop is executed at most 5 times and the whole procedure works in  $O(n^{\omega/2})$ .

## 5 Maximum Versus Perfect Matchings

We now show that the problem of finding a maximum matching can be reduced in time  $O(n^{\omega/2})$  to the problem of finding a perfect matching. The problem is to find the largest subset  $W \subseteq V$ , such that the induced  $G[W]$  has a perfect matching. Notice that this is equivalent to finding the largest subset  $W \subseteq V$ , such that  $A_{W,W}$  is non-singular. The basic idea is to use the nested dissection algorithm. We first show that non-singular submatrices of  $AA^T$  correspond to non-singular submatrices of  $A$  (note that Lemma 15, Theorem 17 and Theorem 18 are all well known facts).

**Lemma 15.** *The matrix  $AA^T$  has the same rank as  $A$ .*

*Proof.* We will prove that  $\ker(A) = \ker(A^T A)$ . Let  $v$  be such that  $(A^T A)v = 0$ . We have

$$0 = v^T (A^T A)v = (v^T A^T)(Av) = (Av)^T (Av),$$

so  $Av = 0$ . □

*Remark 16.* Notice that this proof (and the proof of Lemma 19 later in this section) relies on the fact that  $vv^T = 0$  iff  $v = 0$ . This is not true for finite fields and we will have to deal with this problem in the next section.

We will also need the following classic theorem of Frobenius (see [21])

**Theorem 17 (Frobenius Theorem).** *Let  $A$  be an  $n \times n$  skew-symmetric matrix and let  $X, Y \subseteq \{1, \dots, n\}$  such that  $|X| = |Y| = \text{rank}(A)$ . Then*

$$\det(A_{X,X}) \det(A_{Y,Y}) = (-1)^{|X|} \det^2(A_{X,Y}).$$

Now, we are ready to prove the following

**Theorem 18.** *If  $(AA^T)_{W,W}$  is non-singular and  $|W| = \text{rank}(AA^T)$ , then  $A_{W,W}$  is also non-singular.*

*Proof.* We have  $(AA^T)_{W,W} = A_{W,V} A_{V,W}^T$ , so  $\text{rank}(A_{W,V}) = \text{rank}(AA^T)$ . By Lemma 15, this is equal to  $\text{rank}(A)$ . Let  $A_{W,U}$  be any square submatrix of  $A_{W,V}$  of maximal rank. From Frobenius Theorem it follows that  $A_{W,W}$  also has maximal rank. □

The only question now is, whether  $AA^T$  always has a submatrix  $(AA^T)_{W,W}$  (i.e., a symmetrically placed submatrix) of maximal rank. There are many ways to prove this fact, but we use the one that leads to an algorithm for actually finding this submatrix.

**Lemma 19.** *If  $A$  is any matrix and  $(AA^T)_{i,i} = 0$ , then  $(AA^T)_{i,j} = (AA^T)_{j,i} = 0$  for all  $j$ .*

*Proof.* Let  $e_i$  be the  $i$ -th unit vector. We have

$$0 = (AA^T)_{i,i} = (e_i^T A)(A^T e_i) = (A^T e_i)^T (A^T e_i),$$

so  $A^T e_i = 0$ . But then  $(AA^T)_{i,j} = (e_i^T A)(A^T e_j) = 0$  for any  $j$  and the same for  $(AA^T)_{j,i}$ .  $\square$

**Theorem 20.** *A submatrix  $(AA^T)_{W,W}$  of  $AA^T$  of maximal rank always exists and can be found in time  $O(n^{\omega/2})$  using the nested dissection algorithm.*

*Proof.* We perform the nested dissection algorithm on the matrix  $AA^T$ . At any stage of the computations, the matrix we are working on is of the form  $BB^T$  for some  $B$ . It follows from Lemma 19, that if a diagonal entry we want to eliminate has value zero, then the row and the column corresponding to this entry consist of only zeros. We ignore these and proceed with the elimination. The matrix  $(AA^T)$  we are looking for consists of all non-ignored rows and columns.  $\square$

**Corollary 21.** *For any planar graph  $G = (V, E)$  a largest subset  $W \subseteq V$ , such that  $G[W]$  has a perfect matching, can be found in time  $O(n^{\omega/2})$ .*

We have thus argued that for planar graphs the maximum matching problem can be reduced in time  $O(n^{\omega/2})$  to the perfect matching problem. Since we can solve the latter in time  $O(n^{\omega/2})$ , we can solve the former in the same time.

## 6 Working over a Finite Field

So far, we have shown an algorithm finding a maximum matching in a planar graph using  $O(n^{\omega/2})$  rational arithmetic operations. It can be shown that, with high probability, our algorithm gives the same results if performed using the finite field arithmetic  $\mathbb{Z}_p$  for a randomly chosen prime  $p = \Theta(n^4)$ . Due to space limitations we defer these considerations to the full version of this paper.

**Acknowledgements.** The authors would like to thank their favourite supervisor Krzysztof Diks for numerous useful discussions.

## References

1. Mucha, M., Sankowski, P.: Maximum matchings via gaussian elimination. 45th Annual IEEE Symposium on Foundations of Computer Science, accepted (2004)
2. Edmonds, J.: Paths, trees and flowers. Canadian Journal of Mathematics **17** (1965) 449–467
3. Micali, S., Vazirani, V.V.: An  $o(\sqrt{|V||E|})$  algorithm for finding maximum matching in general graphs. In: Proceedings of the twenty first annual IEEE Symposium on Foundations of Computer Science. (1980) 17–27

4. Blum, N.: A new approach to maximum matching in general graphs. In: Proc. 17th ICALP. Volume 443 of LNCS., Springer-Verlag (1990) 586–597
5. Gabow, H.N., Tarjan, R.E.: Faster scaling algorithms for general graph matching problems. *J. ACM* **38** (1991) 815–853
6. Miller, G.L., Naor, J.: Flow in planar graphs with multiple sources and sinks. In: Proc. 30th IEEE Symp. Foundations of Computer Science. (1989) 112–117
7. Klein, P., Rao, S., Rauch, M., Subramanian, S.: Faster shortest-path algorithms for planar graphs. In: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, ACM Press (1994) 27–37
8. Lovász, L.: On determinants, matchings and random algorithms. In Budach, L., ed.: *Fundamentals of Computation Theory*, Akademie-Verlag (1979) 565–574
9. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: Proceedings of the nineteenth annual ACM conference on Theory of computing, ACM Press (1987) 1–6
10. Strassen, V.: Gaussian elimination is not optimal. *Numerische Mathematik* **13** (1969) 354–356
11. Wilson, D.B.: Determinant algorithms for random planar structures. In: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics (1997) 258–267
12. Tutte, W.T.: The factorization of linear graphs. *J. London Math. Soc.* **22** (1947) 107–111
13. Rabin, M.O., Vazirani, V.V.: Maximum matchings in general graphs through randomization. *Journal of Algorithms* **10** (1989) 557–567
14. Zippel, R.: Probabilistic algorithms for sparse polynomials. In: International Symposium on Symbolic and Algebraic Computation. Volume 72 of LNCS., Berlin, Springer-Verlag (1979) 216–226
15. Schwartz, J.: Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM* **27** (1980) 701–717
16. Bunch, J., Hopcroft, J.: Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation* **28** (1974) 231–236
17. Lipton, R.J., Tarjan, R.E.: A separator theorem for planar graphs. *SIAM J. Applied Math.* (1979) 177–189
18. Lipton, R.J., Rose, D.J., Tarjan, R.: Generalized nested dissection. *SIAM J. Num. Anal.* **16** (1979) 346–358
19. Pan, V.Y., Reif, J.H.: Fast and efficient parallel solution of sparse linear systems. *SIAM J. Comput.* **22** (1993) 1227–1250
20. Khaira, M.S., Miller, G.L., Sheffler, T.J.: Nested dissection: A survey. Technical Report CS-92-106 (1992)
21. Kowalewski, G.: *Einführung in die Determinanten Theorie*. Leipzig Verlag von Veit & Co. (1909)



# Fast Multipoint Evaluation of Bivariate Polynomials

Michael Nüsken and Martin Ziegler\*

University of Paderborn, 33095 Paderborn, GERMANY  
{nuesken,ziegler}@upb.de

**Abstract.** We generalize univariate multipoint evaluation of polynomials of degree  $n$  at sublinear amortized cost per point. More precisely, it is shown how to evaluate a bivariate polynomial  $p$  of maximum degree less than  $n$ , specified by its  $n^2$  coefficients, simultaneously at  $n^2$  given points using a total of  $\mathcal{O}(n^{2.667})$  arithmetic operations. In terms of the input size  $N$  being quadratic in  $n$ , this amounts to an amortized cost of  $\mathcal{O}(N^{0.334})$  per point.

## 1 Introduction

By Horner’s Rule, any polynomial  $p$  of degree less than  $n$  can be evaluated at a given argument  $x$  in  $\mathcal{O}(n)$  arithmetic operations which is optimal for a generic polynomial as proved by Pan (1966), see for example Theorem 6.5 in Bürgisser, Clausen & Shokrollahi (1997).

In order to evaluate  $p$  at *several* points, we might sequentially compute  $p(x_k)$  for  $0 \leq k < n$ . However, regarding that both the input consisting of  $n$  coefficients of  $p$  and  $n$  points  $x_k$  and the output consisting of the  $n$  values  $p(x_k)$  have only linear size, information theory provides no justification for this quadratic total running time. In fact, a more sophisticated algorithm permits to compute all  $p(x_k)$  simultaneously using only  $\mathcal{O}(n \cdot \log^2 n \cdot \log \log n)$  operations. Based on the *Fast Fourier Transform*, the mentioned algorithms and others realize what is known as *Fast Polynomial Arithmetic*. For ease of notation, we use the ‘soft-Oh’ notation, namely  $\mathcal{O}^\sim(f(n)) := \mathcal{O}(f(n)(\log f(n))^{\mathcal{O}(1)})$ . This variant of the usual asymptotic ‘big-Oh’ notation ignores poly-logarithmic factors like  $\log^2 n \cdot \log \log n$ .

**Fact 1.** *Let  $R$  be a commutative ring with one.*

- (i) *Multiplication of univariate polynomials: Suppose we are given polynomials  $p, q \in R[X]$  of degree less than  $n$ , specified by their coefficients. Then we can compute the coefficients of the product polynomial  $p \cdot q \in R[X]$  using  $\mathcal{O}^\sim(n)$  arithmetic operations in  $R$ .*

---

\* Supported by the DFG Research Training Group GK-693 of the Paderborn Institute for Scientific Computation (PaSCo)

- (ii) Multipoint evaluation of a univariate polynomial: Suppose we are given a polynomial  $p \in R[X]$  of degree less than  $n$ , again specified by its coefficients, and points  $x_0, \dots, x_{n-1} \in R$ . Then we can compute the values  $p(x_0), \dots, p(x_{n-1}) \in R$  using  $\mathcal{O}^\sim(n)$  arithmetic operations in  $R$ .
- (iii) Univariate interpolation: Conversely, suppose we are given points  $(x_k, y_k) \in R^2$  for  $0 \leq k < n$  such that  $x_k - x_\ell$  is invertible in  $R$  for all  $k \neq \ell$ . Then we can compute the coefficients of a polynomial  $p \in R[X]$  of degree less than  $n$  such that  $p(x_k) = y_k$ ,  $0 \leq k < n$ , that is, determine the interpolation polynomial to data  $(x_k, y_k)$  using  $\mathcal{O}^\sim(n)$  arithmetic operations in  $R$ .

*Proof.* These results can be found for example in von zur Gathen & Gerhard (2003) including small constants:

- (i) can be done using at most  $63.427 \cdot n \cdot \log_2 n \cdot \log_2 \log_2 n + \mathcal{O}(n \log n)$  arithmetic operations in  $R$  by Theorem 8.23. The essential ingredient is the Fast Fourier Transform. If  $R = \mathbb{C}$  then even  $\frac{9}{2}n \log_2 n + \mathcal{O}(n)$  arithmetic operations suffice. This goes back to Schönhage & Strassen (1971) and Schönhage (1977).

In the following  $M(n)$  denotes the cost of one multiplication of univariate polynomials over  $R$  of degree less than  $n$ .

- (ii) can be done using at most  $\frac{11}{2}M(n) \log_2 n + \mathcal{O}(n \log n)$  operations in  $R$  according to Corollary 10.8. Here, Divide & Conquer provides the final building block. This goes back to Fiduccia (1972).
- (iii) can be done using at most  $\frac{13}{2}M(n) \log_2 n + \mathcal{O}(n \log n)$  operations in  $R$  according to Corollary 10.12. This, too, is completed by Divide & Conquer. The result goes back to Horowitz (1972).

You also find an excellent account of all these in Borodin & Munro (1975).  $\square$

Fast polynomial arithmetic and in particular multipoint evaluation has found many applications in algorithmic number theory (see for example Odlyzko & Schönhage 1988), computer aided geometric design (see for example Lodha & Goldman 1997), and computational physics (see for example Ziegler ???b).

Observe that the above claims apply to the *univariate* case. What about multivariate analogues? Let us for a start consider the bivariate case: A bivariate polynomial  $p \in R[X, Y]$  of *maximum degree*  $\max \deg p := \max \{\deg_X p, \deg_Y p\}$  less than  $n$  has up to  $n^2$  coefficients, one for each monomial  $X^i Y^j$  with  $0 \leq i, j < n$ . Now corresponding to Fact 1, the following questions emerge:

- Question 2.** (i) Multiplication of bivariate polynomials: *Can two given bivariate polynomials of maximum degree less than  $n$  be multiplied within time  $\mathcal{O}^\sim(n^2)$ ?*
- (ii) Multipoint evaluation of a bivariate polynomial: *Can a given bivariate polynomial of maximum degree less than  $n$  be evaluated simultaneously at  $n^2$  arguments in time  $\mathcal{O}^\sim(n^2)$ ?*
  - (iii) Bivariate interpolation: *Given  $n^2$  points  $(x_k, y_k, z_k) \in R^3$ , is there a polynomial  $p \in R[X, Y]$  of maximum degree less than  $n$  such that  $p(x_k, y_k) = z_k$  for all  $0 \leq k < n^2$ ? And, if yes, can we compute it in time  $\mathcal{O}^\sim(n^2)$ ?*

Such issues also arise for instance in connection with fast arithmetic for polynomials over the skew-field of hypercomplex numbers (Ziegler 1997a, Section 3.1).

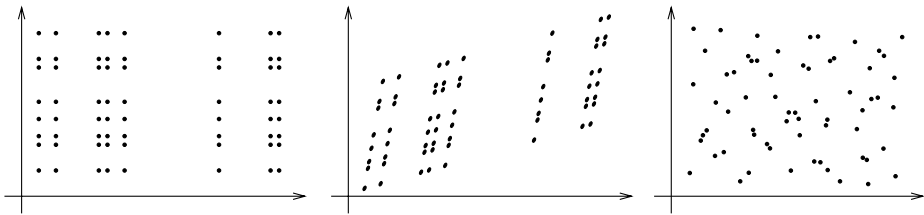
A positive answer to Question 2(i) is achieved by embedding  $p$  and  $q$  into univariate polynomials of degree  $\mathcal{O}(n^2)$  using the *Kronecker substitution*  $Y = X^{2n-1}$ , applying Fact 1(i) to them, and then re-substituting the result to a bivariate polynomial; see for example Corollary 8.28 in von zur Gathen & Gerhard (2003) or Section 1.8 in Bini & Pan (1994).

Note that the first part of (iii) has negative answer for instance whenever the points  $(x_k, y_k)$  are co-linear or, more generally, lie on a curve of small degree: Here, a bivariate polynomial of maximum degree less than  $n$  does not even exist in general.

Addressing (ii), observe that Kronecker substitution is not compatible with evaluation and thus of no direct use for reducing to the univariate case. The methods that yield Fact 1(ii) are not applicable either as they rely on fast polynomial division with remainder which loses many of its nice mathematical and computational properties when passing from the univariate to the bivariate case.

Nevertheless, (ii) does admit a rather immediate positive answer *provided* the arguments  $(x_k, y_k)$ ,  $0 \leq k < n^2$  form a Cartesian  $n \times n$ -grid (also called tensor product grid). Indeed, consider  $p(X, Y) = \sum_{0 \leq j < n} q_j(X)Y^j$  as a polynomial in  $Y$  with coefficients  $q_j$  being univariate polynomials in  $X$ . Then multi-evaluate  $q_j$  at the  $n$  distinct values  $x_k$ : as  $q_j$  has degree less than  $n$ , this takes time  $\mathcal{O}^\sim(n)$  for each  $j$ , adding to a total of  $\mathcal{O}^\sim(n^2)$ . Finally take the  $n$  different univariate polynomials  $p(x_k, Y)$  in  $Y$  of degree less than  $n$  and multi-evaluate each at the  $n$  distinct values  $y_\ell$ : this takes another  $\mathcal{O}^\sim(n^2)$ .

The presumption on the arguments to form a Cartesian grid allows for a slight relaxation in that this grid may be rotated and sheared: Such kind of



**Fig. 1.** Cartesian  $8 \times 8$ -grid, same rotated and sheared; 64 generic points.

affine distortion is easy to detect, reverted to the arguments, and then instead applied to the polynomial  $p$  by transforming its coefficients within time  $\mathcal{O}^\sim(n^2)$ , see Lemma 14 below. The obtained polynomial  $\hat{p}$  can then be evaluated on the now strictly Cartesian grid as described above. However,  $n \times n$  grids, even rotated and sheared ones, form only a zero-set within the  $2n^2$ -dimensional space of all possible configurations of  $n^2$  points. Thus this is a severe restriction.

## 2 Goal and Idea

The big open question and goal of the present work is concerned with fast multipoint evaluation of a multivariate polynomial. As a first step in this direction we consider the bivariate case.

The naïve approach to this problem, namely of sequentially calculating all  $p(x_k, y_k)$ , takes quadratic time each, thus inferring total cost of order  $n^4$ . A first improvement to  $\mathcal{O}^\sim(n^3)$  is based on the simple observation that any  $n$  points in the plane can easily be extended to an  $n \times n$  grid on which, by the above considerations, multipoint evaluation of  $p$  is feasible in time  $\mathcal{O}^\sim(n^2)$ . So we may partition the  $n^2$  arguments into  $n$  blocks of  $n$  points and multi-evaluate  $p$  sequentially on each of them to obtain the following

**Theorem 3.** *Let  $R$  be a commutative ring with one. A bivariate polynomial  $p \in R[X, Y]$  of  $\deg_X(p) < n$  and  $\deg_Y(p) < n$ , given by its coefficients, can be evaluated simultaneously at  $n^2$  given arguments  $(x_k, y_k)$  using at most  $\mathcal{O}(n^3 \cdot \log^2 n \cdot \log \log n)$  arithmetic operations in  $R$ .*

We reduce this softly cubic upper complexity bound to  $\mathcal{O}(n^{2.667})$ . More precisely, by combining fast univariate polynomial arithmetic with fast matrix multiplication we will prove:

**Result 4.** *Let  $\mathbb{K}$  denote an arbitrary field. A bivariate polynomial  $p \in \mathbb{K}[X, Y]$  of  $\deg_X(p) < n$  and  $\deg_Y(p) < m$ , specified by its coefficients, can be evaluated simultaneously at  $N$  given arguments  $(x_k, y_k) \in \mathbb{K}^2$  with pairwise different first coordinates using  $\mathcal{O}((N + nm)m^{\omega_2/2-1+\varepsilon})$  arithmetic operations in  $\mathbb{K}$  for any fixed  $\varepsilon > 0$ .*

Here,  $\omega_2$  denotes the exponent of the multiplication of  $n \times n$ - by rectangular  $n \times n^2$ -matrices, see Section 3. In fact this problem is well-known to admit a much faster solution than naïve  $\mathcal{O}(n^4)$ , the current world record  $\omega_2 < 3.334$  being due to Huang & Pan (1998). By choosing  $m = n$  and  $N = n^2$ , this yields the running time claimed in the abstract.

The general idea underlying Result 4, illustrated for the case of  $n = m$ , is to reduce the bivariate to the univariate case by substituting  $Y$  in  $p(X, Y)$  with the interpolation polynomial  $g(X)$  of degree less than  $n^2$  to data  $(x_k, y_k)$ . It then suffices to multi-evaluate the univariate result  $p(X, g(X))$  at the  $n^2$  arguments  $x_k$ . Obviously, this can only work if such an interpolation polynomial  $g$  is available, that is any two evaluation points  $(x_k, y_k) \neq (x_{k'}, y_{k'})$  differ in their first coordinates,  $x_k \neq x_{k'}$ . However, this condition can be asserted easily later on, see Section 6, so for now assume it is fulfilled.

This naïve substitution leads to a polynomial of degree up to  $\mathcal{O}(n^3)$ . On the other hand, it obviously suffices to obtain  $p(X, g(X))$  modulo the polynomial  $f(X) := \prod_{0 \leq k < n^2} (X - x_k)$  which has degree less than  $n^2$ . The key to efficient bivariate multipoint evaluation is thus an efficient algorithm for this *modular bi-to-univariate composition* problem, presented in Theorem 9.

As we make heavy use of fast matrix multiplication, Section 3 recalls some basic facts, observations, and the state of the art in that field of research. Section 4 formally states the main result of the present work together with two tools (affine substitution and modular composition) which might be interesting on their own, their proofs being postponed to Section 5. Section 6 describes three ways to deal with arguments that do have coinciding first coordinates. Section 7 gives some final remarks.

### 3 Basics on Fast Matrix Multiplication

Recall that, for a field  $\mathbb{K}$ ,  $\omega = \omega(\mathbb{K}) \geq 2$  denotes the *exponent of matrix multiplication*, that is, the least real such that  $m \times m$  matrix multiplication is feasible in asymptotic time  $\mathcal{O}(m^{\omega+\varepsilon})$  for any  $\varepsilon > 0$ ; see for example Chapter 15 in Bürgisser *et al.* (1997). The current world-record due to Coppersmith & Winograd (1990) achieves  $\omega < 2.376$  independent of the ground field  $\mathbb{K}$ . The Notes 12.1 in von zur Gathen & Gerhard (2003) contain a short historical account.

Clearly, a rectangular matrix multiplication of, say,  $m \times m$ -matrices by  $m \times m^t$ -matrices can always be done partitioning into  $m \times m$  square matrices. Yet, in some cases there are better known algorithms than this. We use the notation introduced by Huang & Pan (1998):  $\omega(r, s, t)$  denotes the exponent of the multiplication of  $\lceil m^r \rceil \times \lceil m^s \rceil$ - by  $\lceil m^s \rceil \times \lceil m^t \rceil$ -matrices, that is

$$\omega(r, s, t) = \inf \left\{ \tau \in \mathbb{R} \left| \begin{array}{l} \text{Multiplication of } \lceil m^r \rceil \times \lceil m^s \rceil \text{- by} \\ \lceil m^s \rceil \times \lceil m^t \rceil \text{-matrices can be done} \\ \text{with } \mathcal{O}(m^\tau) \text{ arithmetic operations} \end{array} \right. \right\}.$$

Clearly,  $\omega = \omega(1, 1, 1)$ . We always have

$$\max \{r + s, r + t, s + t\} \leq \omega(r, s, t) \leq r + s + t. \quad (5)$$

Note that  $\omega(r, s, t)$  is in fact invariant under permutation of its arguments.

We collect some known bounds on fast matrix multiplication algorithms.

**Fact 6.** (i)  $\omega = \omega(1, 1, 1) \leq \log_2(7) < 2.8073549221$  (Strassen 1969).

(ii)  $\omega = \omega(1, 1, 1) < 2.3754769128$  (Coppersmith & Winograd 1990).

(iii)  $\omega_2 := \omega(1, 1, 2) < 3.3339532438$  (Huang & Pan 1998).

Partitioning into square matrices only yields  $\omega_2 \leq \omega + 1 < 3.3754769128$ . Bounds for further rectangular matrix multiplications can be also be found in Huang & Pan (1998). It is conjectured that  $\omega = 2$ . Then by partitioning into square blocks also  $\omega(r, s, t)$  touches its lower bound in (5), that is  $\omega(r, s, t) = \max \{r + s, r + t, s + t\}$ . In particular,  $\omega_2 = 3$  then.

We point out that the definition of  $\omega$  and  $\omega(r, s, t)$  refers to arbitrary algebraic computations which furthermore may be non-uniform, that is, use for each matrix size  $m$  a different algorithm. However, closer inspection of Section 15.1 in Bürgisser *et al.* (1997) reveals the following

**Observation 7.** Rectangular matrix multiplication of  $\lceil m^r \rceil \times \lceil m^s \rceil$ - by  $\lceil m^s \rceil \times \lceil m^t \rceil$ -matrices over  $\mathbb{K}$  can be done with  $\mathcal{O}(m^{\omega(r,s,t)+\varepsilon})$  arithmetic operations in  $\mathbb{K}$  by a uniform, bilinear algorithm for any fixed  $\varepsilon$ .

A bilinear computation is a very special kind of algorithm where apart from additions and scalar multiplications only bilinear multiplications occur; see for example Definition 14.7 in Bürgisser *et al.* (1997) for more details. In particular, no divisions are allowed.

## 4 Main Results

Our major contribution concerns bivariate multi-evaluation at arguments  $(x_k, y_k)$  under the condition that their first coordinates  $x_k$  are pairwise distinct. This amounts to a weakened *general position* presumption as is common for instance in Computational Geometry.

For notational convenience, we define ‘ $\mathcal{O}^\approx$ ’ (smooth-Oh) which, in addition to polylogarithmic factors in  $n$ , also ignores factors  $n^\varepsilon$  as long as  $\varepsilon > 0$  can be chosen arbitrarily small. Formally,  $\mathcal{O}^\approx(f(n)) := \bigcap_{\varepsilon > 0} \mathcal{O}(f(n)^{1+\varepsilon})$ . Note that  $\mathcal{O}^\sim(f(n)) \subset \mathcal{O}^\approx(f(n))$ .

**Theorem 8.** Let  $\mathbb{K}$  denote a field. Suppose  $n, m \in \mathbb{N}$ . Given the  $nm$  coefficients of a bivariate polynomial  $p$  with  $\deg_X(p) < n$  and  $\deg_Y(p) < m$  and given  $nm$  points  $(x_k, y_k) \in \mathbb{K}^2$ ,  $0 \leq k < nm$  such that the first coordinates  $x_k$  are pairwise different, we can calculate the  $n$  values  $p(x_k, y_k)$  using  $\mathcal{O}^\approx(nm^{\omega_2/2})$  arithmetic operations over  $\mathbb{K}$ . The algorithm is uniform.

Observe that this yields the first part of Result 4 by performing  $\lceil N/(nm) \rceil$  separate multipoint evaluations at  $nm$  points each. Let us also remark that any further progress in matrix multiplication immediately carries over to our problem. As it is conjectured that  $\omega = 2$  holds, this would lead to bivariate multipoint evaluation within time  $\mathcal{O}^\approx(nm^{1.5})$ .

Our proof of Theorem 8 is based on the following generalization of Brent & Kung’s efficient *univariate* modular composition, see for example Section 12.2 in von zur Gathen & Gerhard (2003), to a certain ‘*bi-to-univariate*’ variant:

**Theorem 9.** Fix a field  $\mathbb{K}$ . Given  $n, m \in \mathbb{N}$ , a bivariate polynomial  $p \in \mathbb{K}[X, Y]$  with  $\deg_X(p) < n$  and  $\deg_Y(p) < m$  and univariate polynomials  $g, f \in \mathbb{K}[X]$  of degree less than  $nm$ , specified by their coefficients. Then  $p(X, g(X)) \bmod f(X)$  can be computed with  $\mathcal{O}^\approx(nm^{\omega_2/2})$  arithmetic operations in  $\mathbb{K}$ .

We remark that *true* bivariate modular computation requires Gröbner basis methods which for complexity reasons are beyond our interest here.

## 5 Proofs

Now we come to the proofs.

**Lemma 10.** *Let  $\mathbb{K}$  denote a field and fix  $t > 0$ .*

- (i) *Let both  $A$  be an  $m \times m$ -matrix and  $B$  an  $m \times m^t$ -matrix whose entries consist of polynomials  $a_{ij}(X), b_{ij}(X) \in \mathbb{K}[X]$  of degree less than  $n$ . Given  $m$  and the  $n \cdot (m^2 + m^t)$  coefficients, we can compute the coefficients of the polynomial entries  $c_{ij}(X)$  of  $C := A \cdot B$  within  $\mathcal{O}^{\approx}(nm^{\omega(1,1,t)})$  arithmetic operations.*
- (ii) *If  $A$  denotes an  $m \times m$  square matrix with polynomial entries of degree less than  $n$  and  $b$  denotes an  $m$ -component vector of polynomials of degree less than  $nm^t$ , then  $(A, b) \mapsto A \cdot b$  is computable within  $\mathcal{O}^{\approx}(nm^{\omega(1,1,t)})$ .*
- (iii) *Let  $p_0, \dots, p_{m-1} \in \mathbb{K}[X, Y]$  denote bivariate polynomials with  $\deg_X(p_i) < n$  and  $\deg_Y(p_i) < m$ , given their  $nm^2$  coefficients, and let furthermore univariate polynomials  $g, f \in \mathbb{K}[X]$  of degree less than  $nm^t$  be given by their coefficients. Then the coefficients of the  $m$  univariate polynomials*

$$p_i(X, g(X)) \bmod f(X)$$

*can be computed with  $\mathcal{O}^{\approx}(nm^{\omega(1,1,t)})$  arithmetic operations.*

*In particular, for  $t = 1$  we have cost  $\mathcal{O}^{\approx}(nm^{\omega}) \subset \mathcal{O}^{\sim}(nm^{2.376})$  and for  $t = 2$  we have cost  $\mathcal{O}^{\approx}(nm^{\omega_2}) \subset \mathcal{O}^{\sim}(nm^{3.334})$ .*

*Proof.* (i) By scalar extension to  $R = \mathbb{K}[X]$  we obtain an algorithm with cost  $\mathcal{O}^{\approx}(m^{\omega(1,1,t)})$  arithmetic operations in  $R$  using Observation 7. For the algorithm scalar extension simply means that we perform any multiplication in  $R$  instead of  $\mathbb{K}$ , multiplications with constants become scalar multiplications. And the cost for one operation in  $R$  is  $\mathcal{O}^{\sim}(n)$  as only polynomials of degree  $n$  have to be multiplied.

- (ii) For each  $j$ ,  $0 \leq j < m$ , decompose the polynomial  $b_j$  of degree less than  $nm^t$  into  $m^t$  polynomials of degree less than  $n$ , that is, write  $b_j(X) = \sum_{0 \leq k < m^t} b_{jk}(X) \cdot X^{kn}$ . The desired polynomial vector is then given by

$$\begin{aligned} (A \cdot b)_i(X) &= \sum_{1 \leq j \leq m} a_{ij}(X) \cdot \left( \sum_{0 \leq k < m^t} b_{jk}(X) \cdot X^{kn} \right) \\ &= \sum_{0 \leq k < m^t} (A \cdot B)_{ik}(X) \cdot X^{kn} \end{aligned} \tag{*}$$

where  $0 \leq i < m$  and  $B := (b_{jk})$  denotes an  $m \times m^t$  matrix of polynomials of degree less than  $n$ . The product  $A \cdot B$  can be computed according to (i) in the claimed running time. Multiplication by  $X^{kn}$  amounts to mere coefficient shifts rather than arithmetic operations. And observing that  $\deg((A \cdot B)_{ik}) < 2n$ , only two consecutive terms in the right hand side of (\*) can overlap. So evaluating this sum amounts to  $m^t$ -fold addition of pairs of polynomials of degree less than  $n$ . Since  $\omega(1, 1, t) \geq 1 + t$  by virtue of (5), this last cost of  $nm^{1+t}$  is also covered by the claimed complexity bound.

(iii) Write each  $p_i$  as a polynomial in  $Y$  with coefficients from  $\mathbb{K}[X]$ , that is

$$p_i(X, Y) = \sum_{0 \leq j < m} q_{ij}(X) \cdot Y^j$$

with all  $q_{ij}(X)$  of degree less than  $n$ . Iteratively compute the  $m$  polynomials  $g_j(X) := g^j(X) \bmod f(X)$ , each of degree less than  $nm^t$ , within time  $\mathcal{O}^\sim(nm^{1+t})$  by fast division with remainder (see for example Theorem 9.6 in von zur Gathen & Gerhard 2003).

By multiplying the matrix  $A := (q_{ij})$  to the vector  $b := (g_j)$  according to (ii), determine the  $m$  polynomials

$$\tilde{p}_i(X) := \sum_{0 \leq j < m} q_{ij}(X) \cdot g_j(X), \quad 0 \leq i < m$$

of degree less than  $n + nm^t$ . For each  $i$  reduce again modulo  $f(X)$  and obtain  $p_i(X, g(X)) \bmod f(X)$  using another  $\mathcal{O}^\sim(nm^{1+t})$  operations. Since  $\omega(1, 1, t) \geq 1 + t$  according to (5), both parts are covered by the claimed running time  $\mathcal{O}^\sim(nm^{\omega(1, 1, t)})$ .  $\square$

Lemma 10 puts us in position to prove Theorem 9.

*Proof (Theorem 9).* Without loss of generality we assume that  $m$  is a square. We use a baby step, giant step strategy: Partition  $p$  into  $\sqrt{m}$  polynomials  $p_i$  of  $\deg_Y(p_i) < \sqrt{m}$ , that is

$$p(X, Y) = \sum_{0 \leq i < \sqrt{m}} p_i(X, Y) \cdot Y^{i\sqrt{m}}.$$

Then apply Lemma 10(iii) with  $t = 2$  and  $m$  replaced by  $\sqrt{m}$  to obtain the  $\sqrt{m}$  polynomials  $\tilde{p}_i(X) := p_i(X, g(X)) \bmod f(X)$  within  $\mathcal{O}^\sim(nm^{\omega_2/2})$  operations. Iteratively determine the  $\sqrt{m}$  polynomials  $\tilde{g}_i(X) := (g(X)^{\sqrt{m}})^i \bmod f(X)$  for  $0 \leq i < \sqrt{m}$  within  $\mathcal{O}^\sim(nm^{3/2})$ . Again,  $\omega_2 \geq 3$  asserts this to remain in the claimed bound. Finally compute

$$p(X, g(X)) \bmod f(X) = \sum_{0 \leq i < \sqrt{m}} \left( \tilde{p}_i(X) \cdot \tilde{g}_i(X) \right) \bmod f(X)$$

using another time  $\mathcal{O}^\sim(nm^{3/2})$ .  $\square$

Based on Theorem 9, the following algorithm realizes the idea expressed in Section 2.

**Algorithm 11.** Generic multipoint evaluation of a bivariate polynomial.

Input: Coefficients of a polynomial  $p \in \mathbb{K}[X, Y]$  of  $\deg_X(p) < n$ ,  $\deg_Y(p) < m$  and points  $(x_k, y_k)$  for  $0 \leq k < nm$  with pairwise different first coordinates  $x_k$ .



Output: The values  $p(x_k, y_k)$  for  $0 \leq k < nm$ .

1. Compute the univariate polynomial  $f(X) := \prod_{0 \leq k < nm} (X - x_k) \in \mathbb{K}[X]$ .
2. Compute an interpolation polynomial  $g \in \mathbb{K}[X]$  of degree less than  $nm$  satisfying  $g(x_k) = y_k$  for all  $0 \leq k < nm$ .
3. Apply Theorem 9 to obtain  $\tilde{p}(X) := p(X, g(X)) \bmod f(X)$ .
4. Multi-evaluate this univariate polynomial  $\tilde{p} \in \mathbb{K}[X]$  of degree less than  $nm$  at the  $nm$  arguments  $x_k$ .
5. Return  $(\tilde{p}(x_k))_{0 \leq k < nm}$ .

*Proof (Theorem 8).* The algorithm is correct by construction.

Step 1 in Algorithm 11 can be done in  $\mathcal{O}^\sim(nm)$  arithmetic operations. As the points  $(x_k, y_k)$  have pairwise different first coordinates, the interpolation problem in Step 2 is solvable and, by virtue of Fact 1(iii), in running time  $\mathcal{O}^\sim(nm)$ . For Step 3 Theorem 9 guarantees running time  $\mathcal{O}^\sim(nm^{\omega_2/2})$ . According to Fact 1(ii), Step 4 is possible within time  $\mathcal{O}^\sim(nm)$ . Summing up, we obtain the claimed running time.  $\square$

## 6 Evaluating at Degenerate Points

Here we indicate how certain fields  $\mathbb{K}$  permit to remove the condition on the evaluation point set imposed in Theorem 8. The idea is to rotate or shear the situation slightly, so that afterwards the point set has pairwise different first coordinates. To this end choose  $\theta \in \mathbb{K}$  arbitrary such that

$$\#\{x_k + \theta y_k \mid 0 \leq k < N\} = N \quad (12)$$

where  $N := nm$  denotes the number of points. Then replace each  $(x_k, y_k)$  by  $(x'_k, y'_k) := (x_k + \theta y_k, y_k)$  and the polynomial  $p$  by  $\hat{p}(X, Y) := p(X - \theta Y, Y)$ . This can be done with  $\mathcal{O}^\sim(n^2 + m^2)$  arithmetic operations, see the more general Lemma 14 below. In any case a perturbation like this might even be a good idea if there are points whose first coordinates are ‘almost equal’ for reasons of numerical stability.

**Lemma 13.** *Let  $\mathbb{K}$  denote a field and  $P = \{(x_k, y_k) \in \mathbb{K}^2 \mid 0 \leq k < N\}$  a collection of  $N$  planar points.*

(i) *If  $\#\mathbb{K} \geq N^2$ , then  $\theta \in \mathbb{K}$  chosen uniformly at random satisfies (12) with probability at least  $\frac{1}{2}$ . Using  $\mathcal{O}(\log N)$  guesses and a total of  $\mathcal{O}(N \cdot \log^2 N)$  operations, we can thus find an appropriate  $\theta$  with high probability.*

*If  $\mathbb{K}$  is even infinite, a single guess almost certainly suffices.*

(ii) *In case  $\mathbb{K} = \mathbb{R}$  or  $\mathbb{K} = \mathbb{C}$ , we can deterministically find an appropriate  $\theta$  in time  $\mathcal{O}(N \cdot \log N)$ .*

(iii) *For a fixed proper extension field  $\mathbb{L}$  of  $\mathbb{K}$ , any  $\theta \in \mathbb{L} \setminus \mathbb{K}$  will do.*

Applying (i) or (ii) together with Lemma 14 affects the running time of Theorem 8 only by the possible change in the  $Y$ -degree. Using (iii) means that all subsequent computations must be performed in  $\mathbb{L}$ . This increases all further costs by no more than an additional constant factor depending on the degree  $[\mathbb{L} : \mathbb{K}]$  only.

*Proof.* (i) Observe that an undesirable  $\theta$  with  $x_k + \theta y_k = x_{k'} + \theta y_{k'}$  implies  $y_k = y_{k'}$  or  $\theta = \frac{x_k - x_{k'}}{y_{k'} - y_k}$ . In the latter case,  $\theta$  is thus uniquely determined by  $\{k, k'\}$ . Since there are at most  $\binom{N}{2} < N^2/2$  such choices  $\{k, k'\}$ , no more than half of the  $\#\mathbb{K} \geq N^2$  possible values of  $\theta$  can be undesirable.

(ii) If  $\mathbb{K} = \mathbb{R}$  choose  $\theta > 0$  such that  $\theta \cdot (y_{\max} - y_{\min}) < \min \{x_k - x_{k'} \mid x_k > x_{k'}\}$ . Such a value  $\theta$  can be found in linear time after sorting the points with respect to their  $x$ -coordinate.

In case  $\mathbb{K} = \mathbb{C}$ , we can do the same with respect to the real parts.

(iii) Simply observe that 1 and  $\theta$  are linearly independent.  $\square$

We now state the already announced

**Lemma 14.** *Let  $R$  be a commutative ring with one. Given  $n \in \mathbb{N}$  and the  $n^2$  coefficients of a polynomial  $p(X, Y) \in R[X, Y]$  of degree less than  $n$  in both  $X$  and  $Y$ . Given furthermore a matrix  $A \in R^{2 \times 2}$  and a vector  $b \in R^2$ . From this, we can compute the coefficients of the affinely transformed polynomial  $p(a_{11}X + a_{12}Y + b_1, a_{21}X + a_{22}Y + b_2)$  using  $\mathcal{O}(n^2 \cdot \log^2 n \cdot \log \log n)$  or  $\mathcal{O}^\sim(n^2)$  arithmetic operations over  $R$ .*

*In the special case  $R = \mathbb{C}$  we can decrease the running time to  $\mathcal{O}(n^2 \log n)$ .*

Lemma 14 straight-forwardly generalizes to  $d$ -variate polynomials and  $d$ -dimensional affine transformations being applicable within time  $\mathcal{O}^\sim(n^d)$  for fixed  $d$ .

*Proof.* We prove this in several steps.

- First we note that, over any commutative ring  $S$  with one, we can compute the *Taylor shift*  $p(X + a)$  of a polynomial  $p \in S[X]$  of degree less than  $n$  by an element  $a \in S$  using  $\mathcal{O}(n \cdot \log^2 n \cdot \log \log n)$  arithmetic operations in  $S$ . There are many solutions for computing the *Taylor shift* of a polynomial. We would like to sketch the divide and conquer solution from Fact 2.1(iv) in von zur Gathen (1990) that works over any ring  $S$ : Precompute all powers  $(X + a)^{2^i}$  for  $0 \leq i \leq \nu := \lfloor \log_2 n \rfloor$ . Then recursively split  $p(X) = p_0(X) + X^{2^\nu} p_1(X)$  with  $\deg p_0 < 2^\nu$  and calculate  $p(X + a) = p_0(X + a) + (X + a)^{2^\nu} p_1(X + a)$ . This amounts to  $\mathcal{O}(n \cdot \log^2 n \cdot \log \log n)$  multiplications in  $S$  and  $\mathcal{O}(n \log n)$  other operations. So we achieve this over any ring  $S$  with  $\mathcal{O}(n \cdot \log^2 n \cdot \log \log n)$  operations.
- Next let  $S = R[Y]$ . Then we can use the previous to compute  $p(X + a, Y)$  or  $p(X + aY, Y)$  for a polynomial  $p \in R[X, Y] = S[X]$  of maximum degree less than  $n$  and an element  $a \in R$ . Using Kronecker substitution for the multiplications in  $R[X, Y]$  this can be done with  $\mathcal{O}(n^2 \cdot \log^2 n \cdot \log \log n)$  arithmetic operations in  $R$ .

- Now we prove the assertion. Scaling is easy:  $p(x, y) \mapsto p(\alpha x, y)$  obviously works within  $\mathcal{O}(n^2)$  steps. Use this and the discussed shifts once or twice.

The solution to Problem 2.6 in Bini & Pan (1994) allows to save a factor  $\log n \cdot \log \log n$  when  $R = S = \mathbb{C}$ .  $\square$

## 7 Conclusion and Further Questions

We lowered the upper complexity bound for multi-evaluating dense bivariate polynomials of degree less than  $n$  with  $n^2$  coefficients at  $n^2$  points with pairwise different first coordinates from naïve  $\mathcal{O}(n^4)$  and  $\mathcal{O}^\sim(n^3)$  to  $\mathcal{O}(n^{2.667})$ . The algorithm is based on fast univariate polynomial arithmetic together with fast matrix multiplication and will immediately benefit from any future improvement of the latter.

With the same technique, evaluation of a trivariate polynomial of maximum degree less than  $n$  at  $n^3$  points can be accelerated from naïve  $\mathcal{O}(n^6)$  to  $\mathcal{O}(n^{4.334})$ .

Regarding that the matrix multiplication method of Huang & Pan (1998) has huge constants hidden in the big-Oh notation, it might in practice be preferable to use either the naïve  $2m^3$  or Strassen's  $4.7m^{2.81}$  algorithm (with some tricks). Applying them to our approach still yields bivariate multipoint evaluation within time  $\mathcal{O}(n^3)$  or  $\mathcal{O}(n^{2.91})$ , respectively, with small big-Oh constants and no hidden factors  $\log n$  in the leading term, that is, faster than Theorem 3.

Further questions to consider are:

- Is it possible to remove even the divisions? This would give a much more stable algorithm and it would also work over many rings.
- As  $\omega \geq 2$ , the above techniques will never get below running times of order  $n^{2.5}$ . Can we achieve an upper complexity bound as close as  $\mathcal{O}^\sim(n^2)$  to the information theoretic lower bound?
- Can multipoint evaluation of trivariate polynomials  $p(X_1, X_2, X_3)$  be performed in time  $\mathcal{O}(n^4)$ ?

**Acknowledgements.** The authors wish to thank David Eppstein (2004) for an inspiring suggestion that finally led to Lemma 13(ii).

## References

- DARIO BINI & VICTOR Y. PAN (1994). *Polynomial and matrix computations*, volume 1 of *Progress in theoretical computer science*. Birkhäuser Verlag, Boston, Basel, Berlin. ISBN 0-8176-3786-9, 3-7643-3786-9.
- A. BORODIN & I. MUNRO (1975). *The Computational Complexity of Algebraic and Numeric Problems*. Number 1 in Theory of computation series. American Elsevier Publishing Company, New York.
- R. P. BRENT & H. T. KUNG (1978). Fast Algorithms for Manipulating Formal Power Series. *Journal of the ACM* **25**(4), 581–595.

- PETER BÜRGISSEER, MICHAEL CLAUSEN & MOHAMMED AMIN SHOKROLLAHI (1997). *Algebraic Complexity Theory*. Number 315 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag.
- DON COPPERSMITH & SHMUEL WINOGRAD (1990). Matrix Multiplication via Arithmetic Progressions. *Journal of Symbolic Computation* **9**, 251–280.
- DAVID EPPSTEIN (2004). Re: Geometry problem: Optimal direction. Known results? Usenet news article. URL <http://mathforum.org/epigone/sci.math.research/slexyaxsle>.
- CHARLES M. FIDUCCIA (1972). Polynomial evaluation via the division algorithm: the fast Fourier transform revisited. In *Proceedings of the Fourth Annual ACM Symposium on the Theory of Computing*, Denver CO, 88–93. ACM Press.
- JOACHIM VON ZUR GATHEN (1990). Functional Decomposition of Polynomials: the Tame Case. *Journal of Symbolic Computation* **9**, 281–299.
- JOACHIM VON ZUR GATHEN & JÜRGEN GERHARD (2003). *Modern Computer Algebra*. Cambridge University Press, Cambridge, UK, 2nd edition. ISBN 0-521-82646-2. URL <http://www-math.upb.de/~aggathen/mca/>. First edition 1999.
- ELLIS HOROWITZ (1972). A fast method for interpolation using preconditioning. *Information Processing Letters* **1**, 157–163.
- XIAOHAN HUANG & VICTOR Y. PAN (1998). Fast Rectangular Matrix Multiplication and Applications. *Journal of Complexity* **14**, 257–299. ISSN 0885-064X.
- SURESH K. LODHA & RON GOLDMAN (1997). A unified approach to evaluation algorithms for multivariate polynomials. *Mathematics of Computation* **66**(220), 1521–1559. ISSN 0025-5718. URL <http://www.ams.org/mcom/1997-66-220/S0025-5718-97-00862-4>.
- A. M. ODLYZKO & ARNOLD SCHÖNHAGE (1988). Fast algorithms for multiple evaluations of the Riemann zeta function. *Transactions of the American Mathematical Society* **309**(2), 797–809.
- V. YA. PAN (1966). О способах вычисления значения многочленов. Успехи Математических Наук **21**(1(127)), 103–134. V. YA. PAN, Methods of computing values of polynomials, *Russian Mathematical Surveys* **21** (1966), 105–136.
- ARNOLD SCHÖNHAGE (1977). Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica* **7**, 395–398.
- ARNOLD SCHÖNHAGE & VOLKER STRASSEN (1971). Schnelle Multiplikation großer Zahlen. *Computing* **7**, 281–292.
- VOLKER STRASSEN (1969). Gaussian Elimination is not Optimal. *Numerische Mathematik* **13**, 354–356.
- MARTIN ZIEGLER (???a). Fast Relative Approximation of Potential Fields. 140–149. URL <http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2748&page=140>.
- MARTIN ZIEGLER (???b). Quasi-optimal Arithmetic for Quaternion Polynomials. 705–715. URL <http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2906&page=705>.

# On Adaptive Integer Sorting

Anna Pagh<sup>1</sup>, Rasmus Pagh<sup>1</sup>, and Mikkel Thorup<sup>2</sup>

<sup>1</sup> IT University of Copenhagen, Rued Langgaardsvej 7, 2300 København S, Denmark  
{annao, pagh}@itu.dk

<sup>2</sup> AT&T Labs — Research, Shannon Laboratory, Florham Park, NJ 07932  
mthorup@research.att.com

**Abstract.** This paper considers integer sorting on a RAM. We show that adaptive sorting of a sequence with  $qn$  inversions is asymptotically equivalent to *multisorting* groups of at most  $q$  keys, and a total of  $n$  keys. Using the recent  $O(n\sqrt{\log \log n})$  expected time sorting of Han and Thorup on each set, we immediately get an adaptive expected sorting time of  $O(n\sqrt{\log \log q})$ . Interestingly, for any positive constant  $\varepsilon$ , we show that multisorting and adaptive inversion sorting can be performed in *linear time* if  $q \leq 2^{(\log n)^{1-\varepsilon}}$ . We also show how to asymptotically improve the running time of any traditional sorting algorithm on a class of inputs much broader than those with few inversions.

## 1 Introduction

Sorting is one of the most researched problems in computer science, both due to the fundamental nature of the problem and due to its practical importance. An important extension to basic sorting algorithms is *adaptive* sorting algorithms, which take advantage of any existing “order” in the input sequence to achieve faster sorting. This is a classical example of the trend in modern computer science to express the complexity of an algorithm not only in terms of the *size* of the problem instance, but also in terms of *properties* of the problem instance. On the practical side, it has been observed that many real-life instances of sorting problems do indeed have structure (see, e.g., [12, p. 54] or [13, p. 126]), and this can be exploited for faster sorting.

A classical measure of order, or rather disorder, is the number of *inversions* in the sequence, that is, the number of pairs of elements that are placed in wrong order compared to the sorted sequence. Using finger search trees one can obtain a sorting algorithm that sorts a sequence of  $n$  elements with  $qn$  inversions in time  $O(n \log q)$ , and there is also a lower bound of  $\Omega(n \log q)$  on the number of comparisons needed to sort such a sequence in the worst case [8].

After Fredman and Willard’s seminal work [6], there has been an extensive interest in algorithms that exploit features of real computers to break comparison-based lower bounds for the case of integer keys stored in single words assuming unit cost operations on words. The model used for such studies is the so-called *word RAM*. The main idea, which goes back to classical techniques such as radix

sort and hashing, is to exploit that one can work with the *representation* of integers. In combination with tabulation and word-level parallelism techniques this has lead to asymptotically very fast sorting algorithms. The currently fastest deterministic and randomized sorting algorithms use  $O(n \log \log n)$  time and  $O(n\sqrt{\log \log n})$  expected time, respectively, and linear space [9,10].

The basic research question in this paper is how well we can exploit few inversions in integer sorting. The most obvious way forward is to use previous reductions from inversion sort to finger searching, but for integer keys, this approach has a matching upper- and lower-bound of  $\Theta(n\sqrt{\log q / \log \log q})$  time [3,4]. However, in this paper, we will get down to time  $O(n \log \log q)$  and  $O(n\sqrt{\log \log q})$  expected time, matching the best bounds for regular sorting. Also, more surprisingly, we will show that we can sort in linear time if for some constant  $\varepsilon > 0$ , we have  $q \leq 2^{(\log n)^{1-\varepsilon}} = n^{1/\log^\varepsilon n}$ .

Our results are achieved via reduction to what we call *multisorting*, which is the problem of sorting several groups of keys with some bound on the size of a group. The output is a list for each group of the keys of the group in sorted order.

To appreciate the power of multisorting, suppose the total number of keys over all groups is  $n$  and that all the keys have values in  $[n]$ . We can now prefix each key with the group it belongs to so that the values become pairs  $(i, x)$  where  $i$  is the group and  $x$  is the original value. Using an array over  $[n]$ , we can sort the keys with the new values in linear time with a 2-pass radix sort. Each group now has its keys sorted in a segment of the whole list, and these segments are easily extracted. Thus, the whole multisort is performed in linear time. If, however, the group size  $q$  is much smaller than  $n$ , say,  $q = 2^{\sqrt{\log n}}$ , we would not know how to sort the groups individually in linear time. This power of multisorting has been used for regular (non-adaptive) sorting in [9].

We show in Section 3 that inversion sorting and multisorting are equivalent in a strong sense, namely, that we can multisort using only one inversion sorting and linear extra time and the other way around.

**Theorem 1.** *The following problems can be reduced to each other in  $O(n)$  time and space on a RAM with word size  $w$ :*

- **Multisorting:** *Sort  $n$   $w$ -bit integer keys in groups, where each group consists of at most  $q$  keys.*
- **Inversion sorting:** *Sort a list of  $n$   $w$ -bit integer keys with a known upper bound  $qn$  on the number of inversions.*

The reduction of inversion sorting to multisorting assumes an upper bound  $qn$  on the number of inversions to be known. However, it is easy to make the algorithm automatically adapt to the number of inversions, with only a constant factor increase in running time. The idea is well known: Perform doubling search for the running time of the inversion sorting algorithm. For a precise statement of this reduction see Section 2.

Multisorting can of course be done by sorting each group separately. If a group has less than  $q$  keys, the recent randomized sorting algorithm by Han

and Thorup [10] sorts it in  $O(\sqrt{\log \log q})$  expected time per key. Thus, for multisorting  $n$  keys in groups of size at most  $q$ , we get an algorithm running in  $O(n\sqrt{\log \log q})$  expected time. More interestingly, in Section 4, we show that we efficiently can multisort much larger groups than we can solve individually. In particular, for any positive constant  $\varepsilon$ , we show that we can multisort in linear time with group size up to  $2^{(\log n)^{1-\varepsilon}}$ . Note that this is close to best possible in that if  $\varepsilon$  was 0, this would be general linear time sorting. On the other hand, if the groups are to be sorted individually, the largest group size we can handle in linear time is only  $2^{(\log n)^{1/2-\varepsilon}}$  using the signature sorting of Andersson et al. [2]. Our efficient new multisorting result is as follows.

**Theorem 2.** *On a RAM, we can multisort  $n$  integer keys in groups of size at most  $q$  in linear space and  $O(n(\log \log n)/(1+\log \log n - \log \log q))$  expected time. In particular, this is linear time for  $q \leq 2^{(\log n)^{1-\varepsilon}}$ , for any constant  $\varepsilon > 0$ .*

Combining with Theorem 1, we get the following adaptive sorting result:

**Corollary 1.** *We can sort  $n$  keys with  $qn$  inversions in linear space and  $O(n(\log \log n)/(1+\log \log n - \log \log q))$  expected time. In particular, this is linear time for  $q \leq 2^{(\log n)^{1-\varepsilon}}$ , for any constant  $\varepsilon > 0$ .*

### 1.1 Our Overlooked Reduction

In a slightly weaker form (c.f. Lemma 1), our reduction from inversion sorting to multisorting is quite simple and works on a comparison-based pointer-machine model of computation. We find it quite surprising that it has been overlooked for so long. Of course, in the classical comparison based model, from a theoretical perspective, the reduction to finger searching is optimal in that it gets down to  $O(n \log q)$  time for  $qn$  inversions.

From a practical perspective, however, a finger search data structure with all its parent, children, and horizontal pointers is pretty slow with a large space overhead. With our reduction, we partition our data set arbitrarily into groups of some limited size, that we sort with our favorite tuned sorting algorithm, be it quick sort, merge sort, or even insertion sort if the groups are small enough.

### 1.2 Other Measures of Disorder

It is folklore knowledge that if a sequence has  $r$  runs, that is,  $r$  increasing and decreasing subsequences, then one can sort in time  $O(nf(r))$ , where  $f(r)$  is the time per operation on a priority queue with  $r$  keys. This time bound is incomparable to bounds in terms of the number of inversions. However, in Section 5 we consider the disorder measure *Osc* [11] which is a natural common generalization of both inversions and runs. Based on *any* non-adaptive sorting algorithm, we get a sorting algorithm that is extra efficient on inputs with low *Osc* disorder. In particular, the obtained algorithm will perform well on inputs with a small number of inversions or runs.

## 2 Eliminating the Need for a Bound on Inversion Number

From Theorem 1 it follows that if we can multisort  $n$  keys with group size at most  $q$  in time  $S(n, q)$ , then we can sort  $n$  keys with up to  $qn$  inversions in  $O(S(n, q))$  time. Within this time-bound, we can also output an error if we have problems because the number of inversions is bigger than  $qn$ . However, in adaptive sorting, no-one tells us the number of inversions. On the other hand, suppose we can generate a sequence of  $q_i$  with  $q_0 = 1$ , and such that  $S(n, q_i) = \Theta(n2^i)$ . Then we can run our algorithm for up to  $nq_i$  inversions for  $i = 0, 1, \dots$  until we successfully produce a sorted list. Even if the number  $qn$  of inversions is unknown, we get a running time of

$$\sum_{i=0,1,\dots,q_{i-1} < q} O(S(n, q_i)) = O(S(n, q))$$

For example, with the function  $S(n, q) = O(n(\log \log n)/(1 + \log \log n - \log \log q))$  in Theorem 2, we can take  $q_i = 2^{\log n^{1-1/2^i}}$ . The time we have to compute  $q_i$  is  $O(S(n, q_i))$ , which is far more than we need.

Very generally, when we have a function like  $S(n, \cdot)$  where we can find a sequence of arguments that asymptotically doubles the value, we call the function *doubleable*. Here, by finding the sequence, we mean that we can construct the relevant parts of it without affecting our overall time and space consumption.

## 3 Equivalence Between Inversion Sorting and Multisorting

In this section we show Theorem 1.

**Lemma 1.** *For any integer  $r \geq 1$  we can reduce, with an additive  $O(n)$  overhead in time, inversion sorting of  $n$  integer keys with at most  $qn$  inversions to performing the following tasks:*

- (i) *Multisorting  $n$  integer keys in groups with at most  $r$  integer keys in each, followed by*
- (ii) *Inversion sorting  $4qn/r$  integer keys with an upper bound  $qn$  on the number of inversions*

*Proof.* The following algorithm will solve the problem as described in the lemma.

1. Divide the  $n$  keys into  $\lceil n/r \rceil$  groups with  $r$  keys in each group, except possibly the last. Denote the groups  $G_1, \dots, G_{\lceil n/r \rceil}$ . For ease of exposition let  $G_{\lceil n/r \rceil+1}$  be an empty group.
2. Multisort the  $n$  keys in the groups of size at most  $r$ , and let  $G_i[1], G_i[2], \dots$  be the sorted order of the keys in group  $i$ .
3. Initialize  $\lceil n/r \rceil + 1$  empty lists denoted  $S_1$ , and  $U_1, \dots, U_{\lceil n/r \rceil}$ .



4. Merge all the groups by looking at two groups at a time. For  $i = 1, \dots, \lceil n/r \rceil$  add the keys in groups  $G_i$  and  $G_{i+1}$ , to  $S_1$ ,  $U_i$ , or  $U_{i+1}$  one at a time as described below. We say that a key in a group  $G_i$  is *unused* if it has not been added to any of the lists  $S_1$  or  $U_i$ .  
While there are any unused keys in  $G_i$  and no more than  $r/2$  used keys in  $G_{i+1}$  we repeatedly find the minimum unused key. In case of a tie the key from  $G_i$  is considered the minimum. The minimum is inserted in  $S_1$  if the previous key inserted in  $S_1$  is not larger, and otherwise it is inserted in  $U_i$  or  $U_{i+1}$  depending on the group it came from. Any remaining keys in  $G_i$  after this are inserted in  $U_i$ .
5. Let  $U$  be the concatenation of the lists  $U_1, \dots, U_{\lceil n/r \rceil}$ .
6. Inversion sort the list  $U$  with  $qn$  as an upper bound on the number of inversions. Denote the resulting sorted list  $S_2$ .
7. Merge the two sorted lists  $S_1$  and  $S_2$ .

It is easy to see that the two merging stages, in steps 4 and 7 of the algorithm, take linear time and that the multisorting in step 2 is as stated in the lemma. Since the ordering of the keys in  $U$  is the same as in the original sequence the number of inversions in  $U$  is bounded by  $qn$ .

What remains to show is that the list  $U$  has size at most  $4qn/r$ . We argue that in all cases in step 4 where a key is added to the list  $U_i$  or  $U_{i+1}$ , this key is involved in at least  $r/2$  inversions. First of all, the minimum key can only be smaller than the last key inserted in  $S_1$  when we have just increased  $i$ , and it must come from  $G_{i+1}$ . But then it must have at least  $r/2$  inversions with unused keys in  $G_i$ . Secondly, whenever we put remaining keys from  $G_i$  in  $U_i$ , each of these has at least  $r/2$  inversions with used keys from  $G_{i+1}$ . Since each key in  $U$  is involved in at least  $r/2$  inversions and there are at most  $qn$  inversions in total, and since each inversion involves two keys, we have an upper bound on the size of  $U$  of  $4qn/r$ .

**Corollary 2.** *On a RAM, inversion sorting of  $n$  integer keys, given an upper bound  $qn$  on the number of inversions, can be reduced to multisorting  $n$  integer keys in groups of size at most  $q$ , using  $O(n)$  time and space.*

*Proof.* Using Lemma 1 with  $r = q \log n$  we get that the time for inversion sorting is  $O(n)$  plus the time for multisorting  $n$  keys in groups of size at most  $q \log n$  and inversion sorting  $4n/\log n$  keys with at most  $qn$  inversions. Sorting  $4n/\log n$  keys can be done in time  $O(n)$  by any optimal comparison-based sorting algorithm.

Multisorting  $n$  keys in groups of size at most  $q \log n$  can be done by multisorting the  $n$  keys in groups of size at most  $q$  and merging the groups into larger ones in  $O(n)$  time. A group of size at most  $q \log n$  is divided up into at most  $\log n$  subgroups of size at most  $q$ . To merge these sorted subgroups into one sorted group, start by inserting the minimum from each subgroup into an atomic heap [7]. Repeatedly remove the minimum from the heap and let it be the next key in the merged group. If the deleted minimum comes from subgroup  $i$ , then insert the next key, in sorted order, from subgroup  $i$  into the atomic heap.

Insertion, deletion, and findmin in an atomic heap with at most  $\log n$  keys takes  $O(1)$  time, after  $O(n)$  preprocessing time [7]. The same atomic heap can be used for all mergings and the total time to merge all groups is hence  $O(n)$ .

**Lemma 2.** *On a RAM with word size  $w$ , multisorting  $n$   $w$ -bit integer keys in groups of size at most  $q$ , can be done by inversion sorting  $n$   $w$ -bit integer keys with an upper bound of  $qn$  inversions plus  $O(n)$  extra time and space.*

*Proof.* The following algorithm will solve the problem as described in the lemma. Assume that the word length  $w$  is at least  $2\lceil\log n\rceil$ . If it is smaller then the sorting can be done in linear time using radix sort.

1. Sort all the keys with respect to the  $2\lceil\log n\rceil$  least significant bits, using radix sort. Keep a reference to the original position as extra information.
2. Distribute the keys in the original groups, using the references, such that each group now is sorted according to the  $2\lceil\log n\rceil$  least significant bits.
3. For each key  $x$ , construct a new  $w$ -bit word  $d_x$ , where the first  $\lceil\log n\rceil$  bits contain the number of the group that the key comes from. The following  $w - 2\lceil\log n\rceil$  bits contain the  $w - 2\lceil\log n\rceil$  most significant bits in  $x$ , and the last  $\lceil\log n\rceil$  bits contain  $x$ 's rank after the sorting and re-distribution in steps 1 and 2.
4. Inversion sort the constructed words from step 3 with  $qn$  as an upper bound on the number of inversions.
5. Construct the multisorted list by scanning the list from step 4 and using the rank (the last  $\lceil\log n\rceil$  bits) to find the original key.

Steps 1, 2, 3 and 5 can clearly be implemented to run in  $O(n)$  time, and provided that  $qn$  is an upper bound on the number of inversions for the sorting in step 4 the time for the algorithm is as stated in the lemma. To see that  $qn$  is an upper bound on the number of inversions, note that since the most significant bits in the words sorted in step 4 encode the group number, there can be no inversions between words in different groups. Since a group contains at most  $q$  keys there are at most  $q$  inversions per word, and  $qn$  is an upper bound on the total number of inversions.

What remains to show is that the algorithm is correct. For two keys  $x$  and  $y$  in the same group and  $x < y$ , it holds that the words  $d_x$  and  $d_y$  constructed in step 3 have the property  $d_x < d_y$ . This together with the fact that the group number is written in the most significant bits in the words sorted in step 4 gives that the list produced by the last step is the input list multisorted as desired.

Theorem 1 follows from Corollary 2 and Lemma 2.

## 4 Fast Multisorting on a RAM

Multisorting can of course be done by sorting one group at a time. In particular, by [9,10] multisorting can be done in linear space and  $O(n\sqrt{\log\log q})$  expected time or deterministic time  $O(n\log\log q)$ .

If  $\log q \leq w^{1/2-\varepsilon}$  for some  $\varepsilon > 0$ , we can use the linear expected time sorting of Andersson et al. [2] on each group. Since  $w \geq \log n$ , this gives linear expected time sorting for  $q \leq 2^{(\log n)^{1/2-\varepsilon}}$ . In this section we show how to exploit the situation of sorting multiple groups to increase the group size to  $2^{(\log n)^{1-\varepsilon}}$ , which is much closer to  $n$ . This gives linear time inversion sorting algorithms for a wide range of parameters.

For our fast randomized sorting algorithm we use the following slight variant from [10] of the signature sorting of Andersson et al. [2].

**Lemma 3.** *With an expected linear-time additive overhead, signature sorting with parameter  $r$  reduces the problem of sorting  $q$  integer keys of  $\ell$  bits to*

- (i) *the problem of sorting  $q$  reduced integer keys of  $4r \log q$  bits each and*
- (ii) *the problem of sorting  $q$  integer fields of  $\ell/r$  bits each.*

*Here (i) has to be solved before (ii).*

Employing ideas from [10], we use the above lemma to prove

**Lemma 4.** *Consider multisorting  $n$  integer keys in groups of size at most  $q$ . In linear expected time and linear space, we can reduce the length of keys by a factor  $(\log n)/(\log q)$ .*

*Proof.* To each of the groups  $S_i$  of size at most  $q$ , we apply the signature sort from Lemma 3 with  $r = (\log n)/(\log q)$ . Then the reduced keys from (i) have  $4r \log q = O(\log n)$  bits.

We are going to sort the reduced keys from all the groups together, but prefixing keys from  $S_i$  with  $i$ . The prefixed keys still have  $O(\log n)$  bits, so we can radix sort them in linear time. From the resulting sorted list we can trivially extract the sorted sublist for each  $S_i$ .

We have now spent linear expected time on reducing the problem to dealing with the fields from Lemma 3 (ii) and their length is only a fraction  $1/r = (\log q)/(\log n)$  of those in the input.

We are now ready to show Theorem 2.

*Proof of Theorem 2:* We want to multisort  $n$  keys in groups of size at most  $q$ , where each key has length  $w$ . We are going to reduce the length of keys using Lemma 4. Using a result of Albers and Hagerup [1], we can sort each group in linear time, when the length of keys is reduced to  $w/(\log q \log \log q)$ . Consequently, the number of times we need to apply Lemma 4 is  $O(\log_{(\log n)/(\log q)}(\log q \log \log q)) = O((\log \log q)/(\log \log n - \log \log q))$ . Since each application takes linear expected time, this gives an expected sorting bound of  $O(n \lceil (\log \log q)/(\log \log n - \log \log q) \rceil)$ . This expression simplifies to the one stated in the theorem. For  $q = 2^{(\log n)^{1-\varepsilon}}$  the time for the algorithm is  $O(n/\varepsilon)$ .  $\square$

Comparing the expression of Theorem 2 with the  $O(n\sqrt{\log \log q})$  bound obtained from [10] on each group, we see that Theorem 2 is better if  $(\log \log n - \log \log q) = \omega(\sqrt{\log \log q})$ , that is, if  $q = 2^{(\log n)^{1-\omega(1/\sqrt{\log \log n})}}$ .

## 5 More General Measures of Disorder on a RAM

This section aims at obtaining a RAM sorting algorithm that performs well both in terms of the number of inversions and the number of runs. This is achieved by bounding the time complexity in terms of the more general *Osc* disorder measure of Levkopoulos and Petersson [11].

For two integers  $a, b$  and a multiset of keys  $S$  define  $osc(a, b, S) = |\{x \in S \mid \min(a, b) < x < \max(a, b)\}|$ . For a sequence of keys  $X = x_1, \dots, x_n$  define

$$Osc(X) = \sum_{i=1}^{n-1} osc(x_i, x_{i+1}, X) .$$

Clearly,  $Osc(X) \leq n^2$ . From [11], we know that  $Osc(X)$  is at most 4 times larger than the number of inversions in  $X$  and at most  $2n$  times larger than the number of runs. Hence, an algorithm that is efficient in terms of  $Osc(X)$  is simultaneously efficient in terms of both the number of inversions and the number of runs.

In comparison-based models the best possible time complexity in terms of  $Osc(X)$  is  $O(n \log(Osc(X)/n))$ . In this section we show that we can replace the logarithms by the time complexity  $t(n)$  of a priority queue, which in turn is the per key cost of non-adaptive sorting of up to  $n$  keys [14]. More precisely,

**Theorem 3.** *Assume that we can sort up to  $q$  integer keys in linear space and time  $O(q \cdot t(q))$  where  $t$  is double-able (c.f. §2). Then we can sort a sequence  $X$  of  $n$  keys in time*

$$O\left(n + \sum_{i=1}^{n-1} t(osc(x_i, x_{i+1}, X))\right) .$$

*In particular, if  $t$  is convex, this is  $O(n \cdot t(Osc(X)/n))$ .*

The rest of this section proves Theorem 3. First we will show how to implement a priority queue with a certain property, and then show how to use this for *Osc* adaptive sorting.

### 5.1 A Last-In-Fast-Out Priority Queue

In this section we describe a priority queue with the property that it is faster to delete keys that were recently inserted into the priority queue than to delete keys that have been in the priority queue for long. The time to delete a key  $x$  will depend on the number of inserts after key  $x$  was inserted. We call this property of a priority queue, which was previously achieved for the cache-oblivious model in [5], *last-in-fast-out*.

We define a **Delete-Insert**( $x, y$ ) operation on a priority queue as a delete operation on key  $x$  followed by an insertion of key  $y$ , where there may be any number of operations in between. For a key  $x$  in a priority queue we denote by  $inserts(x)$  the number of keys inserted in the priority queue after  $x$  is inserted and before  $x$  is deleted from the priority queue.

**Theorem 4.** *Assume that we can sort up to  $n$  integer keys in linear space and time  $O(n \cdot t(n))$  where  $t$  is double-able (c.f. §2). Then there exists a last-in-fast-out priority queue, with capacity for holding at most  $n$  keys at any time, supporting the operations  $\text{Init}()$  in  $O(1)$  time,  $\text{FindMin}()$  in time  $O(1)$ ,  $\text{Insert}(x)$  in time  $O(t(n))$ , and  $\text{Delete-Insert}(x, y)$  in amortized time  $O(t(\text{inserts}(x)))$ , using linear space.*

*Proof.* According to the reduction in [14] the sorting bound assumed in the theorem means that there is a linear space priority queue supporting  $\text{FindMin}()$  in time  $O(1)$ ,  $\text{Delete}(x)$  in time  $O(t(n))$ , and  $\text{Insert}(x)$  in time  $O(t(n))$ , where  $n$  is the number of keys in the priority queue.

Our data structure consists of a number of priority queues, denoted by  $PQ_1, PQ_2, \dots$  with properties as described above. We let  $n_i$  denote an upper bound on the size of  $PQ_i$ , where  $n_i$  satisfies  $\lceil \log(t(n_i)) \rceil = i$ . Since  $t(n)$  is double-able we can compute  $n_i$  in time and space  $O(n_i)$ . The keys in each priority queue  $PQ_i$  are also stored in a list,  $L_i$ , ordered according to the time when the keys were inserted into  $PQ_i$ , i.e., the first key in the list is the most recently inserted key and the last is the first inserted key. There are links between the keys in  $PQ_i$  and  $L_i$ . Each key in the data structure is stored in one of the priority queues, and has a reference to its host. The minimum from each of the priority queues is stored in an atomic heap together with a reference to the priority queue it comes from. The atomic heap supports operations  $\text{Insert}(x)$ ,  $\text{Delete}(x)$ ,  $\text{FindMin}()$ , and  $\text{Find}(x)$  in constant time as long as the number of priority queues is limited to  $\log n$  [7]. There will never be a need for an atomic heap of size greater than  $\log n$ . This is due to the definition of  $n_i$  and because, as we will see in the description of  $\text{Insert}(x)$ , a new priority queue will only be created if all smaller priority queues are full.

$\text{Init}()$  is performed by creating an empty priority queue  $PQ_1$  and an empty atomic heap prepared for a constant number of keys. This can clearly be done in  $O(1)$  time. Note that as the data structure grows we will need a larger atomic heap. Initializing an atomic heap for  $\log n$  keys takes  $O(n)$  time and space, which will not be too expensive, and it is done by standard techniques for increasing the size of a data structure.

$\text{FindMin}()$  is implemented by a  $\text{FindMin}()$  operation in the atomic heap, returning a reference to the priority queue from which the minimum comes, followed by a  $\text{FindMin}()$  in that priority queue, returning a reference to the minimum key.  $\text{FindMin}()$  can be performed in  $O(1)$  time.

$\text{Delete-Insert}(x, y)$  is simply implemented as a  $\text{Delete}(x)$  followed by a  $\text{Insert}(y)$ , hence we do not care about the pairing in the algorithm, only in the analysis.

$\text{Delete}(x)$  is implemented as a  $\text{Delete}(x)$  in the priority queue,  $PQ_i$ , in which the key resides, and a  $\text{Find}(x)$  in the atomic heap. If the key is in the atomic heap, then it means that  $x$  is the minimum in  $PQ_i$ . In this case  $x$  is deleted from the atomic heap. The new minimum in  $PQ_i$  (if any) is found by a  $\text{FindMin}()$  in  $PQ_i$  and this key is inserted into the atomic heap. The time for the operations on

the atomic heap is constant.  $\text{Delete}(x)$  and  $\text{FindMin}()$  in  $PQ_i$  takes time  $O(t(n_i))$  and  $O(1)$  respectively. We will later argue that  $t(n_i) = O(t(\text{inserts}(x)))$ .

When an  $\text{Insert}(x_1)$  operation is performed the key  $x_1$  is inserted into  $PQ_1$  and into  $L_1$  as the first key. This may however make  $PQ_1$  overfull, i.e., the number of keys in  $PQ_1$  exceeds  $n_1$ . If this is the case, the last key in  $L_1$ , denoted  $x_2$ , is removed from  $PQ_1$  and inserted into  $PQ_2$ . If the insertion of  $x_1$  or the deletion of  $x_2$  changes the minimum in  $PQ_1$  then the old minimum is deleted from the atomic heap and the new minimum in  $PQ_1$  is inserted into it. The procedure is repeated for  $i = 2, 3, \dots$  where  $x_i$  is removed from  $PQ_{i-1}$  and inserted into  $PQ_i$ . We choose  $x_i$  as the last key in  $L_{i-1}$ . This is repeated until  $PQ_i$  is not full or there are no more priority queues. In the latter case a new priority queue is created where the key can be inserted. Assume that an insertion ends in  $PQ_i$ . The time for the deletions and insertions in  $PQ_1, \dots, PQ_i$  is  $O(\sum_{j=1}^i t(n_j))$ . Since the sizes of the priority queues grows in such a way that the time for an insertion roughly doubles from  $PQ_j$  to  $PQ_{j+1}$  this sum is bounded by  $O(t(n_i))$ . All other operations during an insert takes less time. In the worst case we do not find a priority queue that is not full until we reach the last one. Hence, the time for insert is  $O(t(n))$ .

Now it is easy to see that if key  $x$  is stored in  $PQ_i$  then there must have been at least  $n_{i-1}$  inserts after  $x$  was inserted. The bound for the delete part of  $\text{Delete-Insert}(x, y)$  follows since  $t(n_{i-1}) = \Theta(t(n_i))$ .

To show that the insert part of a  $\text{Delete-Insert}(x, y)$  operation only increases the time by a constant factor we use a potential argument. The delete operation will pay for the insert. For the sake of argument say that whenever a key  $x$  is deleted from  $PQ_i$  the delete operation pays the double amount of time compared to what it uses and leaves  $O(t(n_i))$  time for the insertion of a key in its place. Whenever a key  $y$  is inserted into the data structure, the first non-full priority queue  $PQ_i$  is the place where the insertion stops. This insertion is paid for by the saved potential, unless the insertion does not fill the place of a deleted key. This only happens so many times as the number of  $\text{Insert}(x)$  operations not coupled with a deletion. Hence, the total extra time is  $O(t(n))$  for each such insertion. The amortized bound on  $\text{Delete-Insert}(x, y)$  follows.

The space usage of the data structure is clearly bounded by  $O(n)$ .

## 5.2 Our General Adaptive Sorting Algorithm

**Theorem 5.** *Suppose that we have a linear space last-in-fast-out priority queue for integer keys, with amortized time complexity  $O(t(n))$  for insertion and  $O(t(\text{inserts}(x)))$  for deleting the minimum  $x$  followed by an insertion. Also, suppose  $t$  is non-decreasing and double-able, and that  $t(n) = O(\log n)$ . Then there is an integer sorting algorithm that uses linear space and on input  $X = x_1, \dots, x_n$  uses time*

$$O\left(n + \sum_{i=1}^{n-1} t(\text{osc}(x_i, x_{i+1}, X))\right).$$

*In particular, if  $t$  is convex, this is  $O(n \cdot t(\text{Osc}(X)/n))$ .*

*Proof.* Initially we split the input into  $O(n/\log n)$  groups of  $\Theta(\log n)$  keys, and sort each group in linear time using an atomic heap. Call the resulting list  $X'$ . By convexity of  $t$ , and since  $t(n) = O(\log n)$ , the sum  $\sum_{i=1}^{n-1} t(\text{osc}(x'_i, x'_{i+1}, X'))$  is larger than the sum in the theorem by at most  $O(n)$ . To sort  $X'$  we first put the minimum of each group into the last-in-fast-out priority queue in  $O(n)$  time. We then repeatedly remove the minimum from the priority queue, and insert the next key from the same group, if any. The total time for reporting the first key from each group is clearly  $O(n)$ . Insertion of a key  $x'_i$  that is not first in a group is amortized for free and the deletion cost is  $O(t(\text{inserts}(x'_i))) = O(t(\text{osc}(x'_{i-1}, x'_i, X')))$ . Summing over all keys we get the desired sum.

Theorems 4 and 5 together imply Theorem 3. As an application, using the  $O(n\sqrt{\log \log n})$  expected time sorting from [10], we get that a sequence  $X$  of  $n$  keys can be sorted in expected time  $O(n\sqrt{\log \log (\text{Osc}(X)/n)})$ .

## 6 Final Remarks

An anonymous reviewer has pointed out an alternative proof of Theorem 1. The main insight is that for a sequence of  $n$  keys with  $qn$  inversions, there is a comparison-based linear time algorithm that splits the keys into a sequence of groups of size at most  $q$  such that no key in a group is smaller than any key in a previous group. In other words, all that remains to sort the entire key set is to multisort the groups.

The groups can be maintained incrementally, inserting keys in the order they appear in the input. Initially we have one group consisting of the first  $q$  keys. We maintain the invariant that all groups contain between  $q/2$  and  $q$  keys. For each group, the minimum key and the group size is maintained. The group of a key  $x$  is found by linear search for the largest minimum key no larger than  $x$ . The total time for these searches is  $O(n)$  since each search step can be associated with at least  $q/2$  inversions. If the size of the group in which  $x$  is to be inserted is  $q$ , we split the group (including  $x$ ) into two groups, using a linear time median finding algorithm. The time for these splittings is constant amortized per inserted key.

## References

1. S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Inf. Comput.*, 136:25–51, 1997.
2. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *J. Comp. Syst. Sc.*, 57:74–93, 1998. Announced at STOC'95.
3. A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. 32<sup>nd</sup> STOC*, pages 335–342, 2000.
4. P. Beame and F. Fich. Optimal bounds for the predecessor problem. In *Proc. 31<sup>st</sup> STOC*, pages 295–304, 1999.
5. G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. 13<sup>th</sup> ISAAC*, volume 2518 of *LNCS*, pages 219–228. 2002.

6. M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47:424–436, 1993. Announced at STOC'90.
7. M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48:533–551, 1994.
8. L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proc. 9<sup>th</sup> STOC*, pages 49–60. 1977.
9. Y. Han. Deterministic sorting in  $O(n \log \log n)$  time and linear space. In *Proc. 34<sup>th</sup> STOC*, pages 602–608, 2002.
10. Y. Han and M. Thorup. Integer sorting in  $O(n\sqrt{\log \log n})$  expected time and linear space. In *Proc. 43<sup>rd</sup> FOCS*, pages 135–144, 2002.
11. C. Levkopoulos and O. Petersson. Adaptive heapsort. *J. Algorithms*, 14(3):395–413, 1993.
12. K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, 1984.
13. R. Sedgewick. *Quicksort*. Garland Pub. Co., New York, 1980.
14. M. Thorup. Equivalence between priority queues and sorting. In *Proc. 43<sup>rd</sup> FOCS*, pages 125–134, 2002.



# Tiling a Polygon with Two Kinds of Rectangles

## (Extended Abstract)

Eric Rémila<sup>1,2</sup>

<sup>1</sup> Laboratoire de l'Informatique du Parallélisme  
UMR 5668 CNRS-INRIA-ENS Lyon-Univ. Lyon 1  
46 allée d'Italie, 69364 Lyon Cedex 07, France

<sup>2</sup> Groupe de Recherche en Informatique et Mathématiques Appliquées  
IUT Roanne (Univ. St-Etienne), 20 avenue de Paris 42334 Roanne Cedex, France  
`Eric.Remila@ens-lyon.fr`

**Abstract.** We fix two rectangles with integer dimensions. We give a quadratic time algorithm which, given a polygon  $F$  as input, produces a tiling of  $F$  with translated copies of our rectangles (or indicates that there is no tiling). Moreover, we prove that any pair of tilings can be linked by a sequence of local transformations of tilings, called flips. This study is based on the use of J. H. Conway's tiling groups and extends the results of C. Kenyon and R. Kenyon (limited to the case when each rectangle has a side of length 1).

## 1 Introduction

In 1990, J. H. Conway and J. C. Lagarias [3] introduce the notion of *tiling group*. They create a way to study tiling problems with an algebraic approach. This work has been prolonged by W. P. Thurston [16], who introduces the notion of *height function* (also introduced independently in the statistical physics literature (see [2] for a review)). The tools cited above have been used and generalized by different authors [5], [6], [7], [9], [12], [13], [14], [15] and some very strong algorithmic and structural results about tilings have been obtained.

In this paper, we consider the problem of tiling a rectilinear polygon (i. e. a not necessarily convex (but simply connected) polygon whose sides consist of horizontal and vertical lines) with rectangular tiles. This problem was previously considered by C. Kenyon and R. Kenyon [5]. These authors obtained a linear time algorithm for tiling a rectilinear polygon by translates of  $m \times 1$  and  $1 \times n'$  rectangles,  $m$  and  $n'$  denoting positive fixed integers. For this, they used Conway-Lagarias type tiling groups, finding an appropriate generalization of “height function” as used by W. P. Thurston for their groups. They also considered tiling using  $2 \times 3$  and  $3 \times 2$  rectangles, where it was asserted that a quadratic time algorithm exists (a brief sketch of an approach without full details is given in the paper). It is also claimed that the method can be extended to the case of tilings by  $k \times l$  and  $l \times k$  rectangles, for any pair  $(k, l)$  of positive integers.

The present paper generalizes the previous paper, considering tiling by two types of tiles, which are translates (we do not allow rotations) of  $m \times n$  and  $m' \times n'$  rectangles, for an arbitrary fixed sequence  $(m, m', n, n')$  of positive integers. Our analysis also uses

Conway-Lagarias type tiling groups. The key new idea needed for the general case is in Proposition 2: we introduce a new technique to reduce the problem by cutting the polygon, using algebraic arguments.

The main results of the paper are:

- the *flip connectivity* (Theorem 1): any pair of tilings of a same polygon can be linked by a sequence of local transformations of tilings, called flips,
- a decision algorithm for tileability of a (rectilinear) polygon  $P$  that is quadratic in the area  $(A(P))$  (Theorem 2).

An important restriction in this work is the fact that tiled regions are assumed no containing interior holes. We recall that it is NP-complete to decide if a general connected region (with holes allowed) is tileable by translates of  $m \times 1$  and  $1 \times n'$  rectangles, with  $m$  and  $n'$  being integers such that  $m \geq 3$  and  $n' \geq 2$  [1].

The paper is divided as follows: we first give the definitions and basic concepts (section 2). Afterwards, we explain our new techniques to cut tilings (section 3) and we show how to combine them with the machinery of height functions (section 4) to get a quadratic tiling algorithm and prove the flip connectivity.

## 2 Basic Tools

**The Square Lattice.** Let  $\Lambda$  be the square lattice of the Euclidean plane  $\mathbb{R}^2$ . A *cell* of  $\Lambda$  is a square of unit sides whose vertices have integer coordinates. The *vertices* and *edges* of  $\Lambda$  are those of the cells of  $\Lambda$ . Two vertices are *neighbors* if they are linked by an edge.

A *path*  $P$  of  $\Lambda$  is a finite sequence of vertices such that two consecutive vertices are neighbors. A *corner* of a path  $P$  is the origin or final vertex, or a vertex  $v$  of  $P$  which is not equal to the center of the line segment linking the predecessor and the successor of  $v$  in  $P$ . A corner  $v$  is *weak* if its predecessor and its successor are equal, otherwise the corner is *strong* (the origin and final vertices of the path are strong corners). A path can be defined by the sequence  $(v_0, v_1, \dots, v_p)$  of its corners. In this paper, we encode paths by this way. A *move* of a path is the subpath linking two consecutive corners. We have two kinds of moves: the horizontal and vertical ones. The *algebraic length* of a horizontal (respectively vertical) move from  $v_i = (x_i, y_i)$  to  $v_{i+1} = (x_{i+1}, y_{i+1})$  is the difference  $x_{i+1} - x_i$  (respectively  $y_{i+1} - y_i$ ).

A (finite) figure  $F$  of  $\Lambda$  is a (finite) union of (closed) cells of  $\Lambda$ . Its boundary is denoted by  $\delta F$ . The *lower boundary* is the union of open horizontal line segments  $]v, v'[,$  included in  $\delta F$ , such that the interior of  $F$  is above  $]v, v'[,$  (i. e. more formally,  $F$  contains a rectangle whose bottom side is  $]v, v'[,$ ). One defines in a similar way the *left, right, and upper boundaries* of  $F$ . A figure  $F$  is *simply connected* if  $F$  and its complement  $\mathbb{R}^2 \setminus F$  both are connected. A finite simply connected figure  $F$  is called a *polygon* of  $\Lambda$ . The boundary of a polygon  $F$  canonically induces a cycle in  $\Lambda$ , which is called the boundary cycle of  $F$ . In this paper we only consider polygons, except when explicitly stated otherwise.

**Tilings.** We fix a pair of  $\{R, R'\}$  of rectangles with vertical and horizontal sides and integer dimensions. The *width* (i. e. horizontal dimension) of  $R$  is denoted by  $m$ , the width of  $R'$  is denoted by  $m'$ ; the *height* (i. e. vertical dimension) of  $R$  is denoted by  $n$ , the height of  $R'$  is denoted by  $n'$ . An  $R$ -tile (respectively  $R'$ -tile) is a translated copy of the rectangle  $R$  (respectively  $R'$ ). A *tiling*  $T$  of a figure  $F$  is a set of tiles, with pairwise disjoint interiors (i. e. there is no overlap), such that the union of the tiles of  $T$  equals  $F$  (i. e. there is no gap). A tile  $t$  is *cut* by a path  $P$  if there exists an open line segment  $]v, v'[,$  linking two consecutive vertices of  $P$ , included in the interior part of the tile  $t$ . The subgraph  $G_T = (V_T, E_T)$  of  $\Lambda$  such that  $V_T$  (respectively  $E_T$ ) is the set (respectively edges) of vertices of  $\Lambda$  which are on the boundary of a tile of  $T$ . Obviously, the set  $V_T$  (respectively  $E_T$ ) contains all the vertices (respectively edges) of  $\Lambda$  which are in  $\delta F$ .

**The Preprocessing.** The problem of tiling can easily be reduced to the case when  $\gcd(m, m') = \gcd(n, n') = 1$  as follows: consider the only tiling  $\Theta$  of the figure  $F$  with translated copies of a  $\gcd(m, m') \times \gcd(n, n')$  rectangles, called *bricks*. The tiling  $\Theta$  is a refinement of any "classical" tiling  $T$  with  $R$ -tiles and  $R'$ -tiles, i. e. each brick of  $\Theta$  is included in a tile of  $T$ .

Two bricks  $\theta$  and  $\theta'$  of  $\Theta$  are called *equivalent* if they respectively are the first and last elements of a finite sequence of bricks such that two consecutive bricks of the sequence share a whole side. Remark that two bricks included in a same tile of  $T$  are equivalent (i. e. the boundary of an equivalence class cuts no tile of  $T$ ), and the union of all the bricks of an equivalence class form a polygon (if such a union had a hole, then a brick placed on a corner of the hole would create a contradiction). Thus, the problem can be simplified as follows: We first compute the equivalence classes (this can easily be done in a linear time in the area of the polygon). Afterwards, we have to study tilings on each equivalence class. But, since in each class, bricks are equivalent, we can change the units in order to consider a brick as the new unit rectangle.

This transformation makes us consider that  $\gcd(m, m') = \gcd(n, n') = 1$ . In the rest of the paper, we will assume that this simplification has been done and thus  $\gcd(m, m') = \gcd(n, n') = 1$ .

### 3 Tiling Groups and Cut Lines

The two following sections are devoted to the main case, called the *non-degenerate case*, when we have  $m > 1, n > 1, m' > 1$  and  $n' > 1$ .

We introduce the groups  $G_1 = \mathbb{Z}/m\mathbb{Z} * \mathbb{Z}/n'\mathbb{Z}$ , i. e. the free product (see for example [8] for details about group theory) of groups  $\mathbb{Z}/m\mathbb{Z}$  and  $\mathbb{Z}/n'\mathbb{Z}$ , and  $G_2 = \mathbb{Z}/m'\mathbb{Z} * \mathbb{Z}/n\mathbb{Z}$ . We only give properties and notations for  $G_1$ , since they are symmetric for  $G_2$ . We recall that there exists a canonical surjection (denoted by  $s_1$ ) from  $G_1$  to the Cartesian product  $\mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/n'\mathbb{Z}$ , and that each element  $g$  of  $G_1$  can be written in a unique way:  $g = \prod_{i=1}^p g_i$ , with the following properties,

- for each integer  $i$  such that  $1 \leq i \leq p$ ,  $g_i$  is element of the set  $(\mathbb{Z}/m\mathbb{Z})^* \cup (\mathbb{Z}/n'\mathbb{Z})^*$ ,
- for each integer  $i$  such that  $1 \leq i < p$ ,  $g_i$  is element of  $\mathbb{Z}/m\mathbb{Z}$  if and only if  $g_{i+1}$  is element of  $\mathbb{Z}/n'\mathbb{Z}$ .

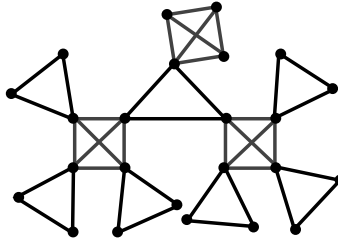


Fig. 1. The graph of  $G_1$  (with  $m = 4$  and  $n' = 3$ ).

This expression is called the *canonical expression* of  $g$  and the integer  $p$  is called the *length* of  $g$  (denoted by  $l(g)$ ). By this way, an order relation is created on  $G_1$ : we say that  $g \leq g'$  if the canonical expression of  $g$  is a prefix of  $g'$ .

Two distinct elements  $g$  and  $g'$  of  $G_1$  are *H-equivalent* if there exists an element  $x$  of  $\mathbb{Z}/m\mathbb{Z}$  such that  $g' = gx$ , they are *V-equivalent* if there exists an element  $y$  of  $\mathbb{Z}/n'\mathbb{Z}$  such that  $g' = gy$ , they are *neighbors* if they are H-equivalent or V-equivalent. Each element of  $G_1$  is exactly the intersection of its H-equivalence class and its V-equivalence class.

The *graph of  $G_1$*  is the symmetric graph whose vertices are elements of  $G_1$ , and for each pair  $(g, g')$  of distinct elements of  $G_1$ ,  $(g, g')$  is an edge if and only if  $g$  and  $g'$  are neighbors<sup>1</sup>.

Remark that each path of this graph linking an element  $g$  to an element  $g'$  has to pass through each element  $g''$  such that  $1_{G_1} \leq g^{-1}g'' \leq g^{-1}g'$ . Informally, we can say that the structure of graphs elements is an infinite tree of equivalent classes.

**Tiling Function.** Let  $P = (v_1, \dots, v_p)$  (according to our convention, the vertices of the sequence are the corners of  $P$ ). This path canonically induces an element  $g_P$  of  $G_1$  as follows: each horizontal (respectively vertical) move gives an element of  $\mathbb{Z}/m\mathbb{Z}$  (respectively  $\mathbb{Z}/n'\mathbb{Z}$ ) equal to its algebraic length; the element  $g_P$  is obtained successively making the product of elements given by successive moves. Notice that any path describing the contour of a tile induces the unit element  $1_{G_1}$ . Thus, tilings can be encoded using the graphs  $G_1$  and  $G_2$ , as we see below.

**Definition 1. (tiling function)** Let  $T$  be a tiling of a polygon  $F$ . A tiling function induced by  $T$  is a mapping  $f_{1,T}$  from the set  $V_T$  to  $G_1$  such that, for each edge  $[v = (x, y), v' = (x', y')]$  cutting no tile of  $T$ :

- if  $x = x'$ , then  $f_{1,T}(v') = f_{1,T}(v)(y' - y)$ , with  $y' - y$  seen as an element of  $\mathbb{Z}/n'\mathbb{Z}$ ,
- if  $y = y'$ , then  $f_{1,T}(v') = f_{1,T}(v)(x' - x)$ , with  $x' - x$  seen as an element of  $\mathbb{Z}/m\mathbb{Z}$ .

**Proposition 1.** (J. H. Conway) let  $F$  be a figure,  $T$  be a tiling of  $F$ ,  $v_0$  be a vertex of the boundary of  $F$  and  $g_0$  be a vertex of  $G_1$ . If  $F$  is connected (respectively is a polygon),

<sup>1</sup> In the language of group theory, the graph of elements is the Cayley graph of  $G_1$  when the set of generators is  $(\mathbb{Z}/m\mathbb{Z})^* \cup (\mathbb{Z}/n'\mathbb{Z})^*$  and labels have been removed.

then there exists at most one (respectively exactly one) tiling function  $f_{1,T}$  induced by  $T$  such that  $f_{1,T}(v_0) = g_0$ .

We will now study tilings of a polygon  $F$ . For each tiling  $T$ , we use the unique tiling function  $f_{1,T}$  such that  $f_{1,T}(O)$  is the unit element of  $G_1$ , where  $O$  denotes a fixed lower left corner of the boundary of  $F$ . Remark that, for each vertex  $v$  on  $\delta F$ ,  $f_{1,T}(v)$  does not depend on the tiling  $T$  (which allows to remove the tiling index and write  $f_1(v)$ ) and can be computed without the knowledge of  $T$ , following the boundary of  $F$ . We also note that, for each vertex  $v$  of  $V_T$ , the value  $s_1 \circ f_{1,T}(v)$  is the pair of coordinates of  $v$  (respectively modulo  $m$  and modulo  $n'$ ), and, consequently, does not depend on  $T$ .

### Cutting Using Algebraic Constraints

**Definition 2.** A special path  $P$  of  $F$  is a path defined by a sequence  $(v_1, \dots, v_p)$  of successive corners such that:

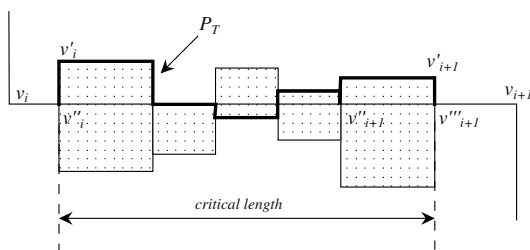
- the vertex  $v_1$  is on the lower boundary of  $F$ , the vertex  $v_p$  is on the boundary of  $F$ , and for each integer  $i$  such that  $1 \leq i < p$ , the line segment  $[v_i, v_{i+1}]$  is included in  $F$ ,
- for each odd integer  $i$  such that  $1 \leq i < p$ , there exists a non-null integer  $k_i$  such that  $-m' < k_i < m'$  and  $v_{i+1} = v_i + (k_i m, 0)$ ,
- for each even integer  $i$  such that  $1 < i < p$ , there exists a non-null integer  $k_i$  such that  $-n < k_i < n$  such that  $v_{i+1} = v_i + (0, k_i n')$ ,
- the element  $(f_2(v_0))^{-1} f_2(v_p)$  is the element  $g_P$  of  $G_2$  induced by the path  $P$ , i. e.  $\prod_{i=1}^{p-1} r_i k_i$ , with  $r_i = n'$  for  $i$  even and  $r_i = m$  for  $i$  odd.

The notion of special path with the proposition below form the main new idea of this paper. There exists a easier proof [4] in the particular case when  $m' = n = 2$  since the graph of  $G_2$  has a linear structure.

**Proposition 2.** Let  $T$  be a tiling and  $P$  be a special path of a polygon  $F$ . The path  $P$  cuts no tile of  $T$ .

*Proof.* We first construct from  $P$  another path  $P_T$  as follows: for each tile  $t$  which is cut by  $P$ , let  $P_{t,cross}$  denote the sub-path of  $P$  which crosses  $t$ . The path  $P_T$  is obtained replacing in  $P$  each sub-path  $P_{t,cross}$  by the shortest path  $P_{t,surround}$ , on the boundary of  $t$ , with the same endpoints as  $P_{t,cross}$  (if two paths are possible, any convention is possible; one can state, for example, that  $t$  is surrounded by left). Notice that  $P_{t,surround}$  is formed from two or three moves. Informally, The path  $P_T$  is an approximation of  $P$  which cuts no tile of  $T$ .

Let  $i$  be an integer such that  $1 \leq i \leq p$ . We define the vertex  $v'_i$  such that,  $v'_i$  is the last *strong corner* of  $P_T$  such that  $f_{2,T}(v'_i)$  is equal to the element  $g_{p_i}$  induced by the sub-path of  $P$  from  $v_0$  to  $v_i$  (i. e.  $g_{p_i} = \prod_{j=0}^i r_j n_j$  (with  $r_j = n'$  for  $j$  even and  $r_j = m$  for  $j$  odd)). This corner exists because of the structural properties of the graphs of elements and classes of  $G_2$  (especially the tree structure of the graph of classes). The point  $v''_i$  is defined as the last point reached in  $P_T$ , before  $v'_i$  which also is in  $P$  (possibly  $v''_i = v'_i$ ; see figure 2 for the notations). We claim the following fact:



**Fig. 2.** The main argument used in the proof of Proposition 2. The critical length has to be equal (modulo  $m'$ ) to the length of the line segment  $[v_i, v_{i+1}]$  and, on the other hand, this length is the sum of widths of tiles.

**Fact:** The vertex  $v_i$  is before  $v''_i$  in  $P$  and, if, moreover,  $v_i = v'_i$  then no tile of  $T$  is cut by  $P$  before this vertex.

The above fact is proved by induction. The result is obviously true for  $i = 1$ . Now, assume that it also holds for a fixed integer  $i$  such that  $1 \leq i < p$ .

If  $v_{i+1}$  is before  $v''_i$  on  $P$ , the result is also obvious, since  $v'_i$  is before  $v''_{i+1}$ . Moreover, remark that, for each vertex  $v$ ,  $s_2 \circ f_{2,T}(v'_i)$  is equal to the pair of coordinates of  $v$  (respectively modulo  $m'$  and  $n$ ), thus  $s_2 \circ f_{2,T}(v'_i) \neq s_2 \circ f_{2,T}(v_{i+1})$ , because of the first component. On the other hand we have  $s_2 \circ f_{2,T}(v'_i) = s_2 \circ f_{2,T}(v_i)$ . Thus  $v'_i \neq v_{i+1}$ , which yields that  $v_{i+1} \neq v'_{i+1}$ .

Otherwise, by induction hypothesis, the point  $v''_i$  is on the line segment  $[v_i, v_{i+1}]$ . We only treat the case when  $i$  is odd and  $k_i$  is positive, since the proof is similar in the other cases.

Since  $v'_i$  is a strong corner of  $P_T$ ,  $v'_i$  is on the left side of a tile  $t_1$  used to define  $P_T$ . The vertex  $v''_i$  is also on the left side of  $t_1$ , thus they have the same horizontal coordinate. Assume that  $v''_{i+1}$  is before  $v_{i+1}$  in  $P$ . In this case,  $v'_{i+1}$  is a right vertex of a tile of  $T$ , and the vertical line passing through  $v'_{i+1}$  cuts the line segment  $[v_i, v_{i+1}]$  in a vertex  $v'''_{i+1}$ .

Now consider the line segment  $[v''_i, v'''_{i+1}]$ . This segment is not reduced to a vertex (since  $v'_i$  and  $v'_{i+1}$  have not the same horizontal component modulo  $m'$ ). Moreover, there exists a sequence  $(t_1, t_2, \dots, t_q)$  of tiles of  $T$  such that the left side of  $t_1$  contains  $v'_i$  and  $v''_i$ , the right side of  $t_q$  contains  $v'_{i+1}$  and  $v'''_{i+1}$ , and for each integer  $j$  such that  $0 \leq j < q$ , the intersection of the right side of  $t_j$  and the left side of  $t_{j+1}$  is a line segment not reduced to a single point. Let  $\mu$  (respectively  $\mu'$ ) denote the number of  $R$ -tiles (respectively  $R'$ -tiles) on the sequence  $(t_1, t_2, \dots, t_q)$ . We have:

$$\mu m + \mu' m' \leq k_i m \quad (1)$$

and, moreover, from the definition of  $v''_i$  and  $v'''_{i+1}$ , using the horizontal coordinates:

$$\mu m + \mu' m' = k_i m (\text{modulo } [m']) \quad (2)$$

Thus, we necessarily have  $(\mu, \mu') = (k_i, 0)$  (since  $\gcd(m, m') = 1$ , the equality (2) gives that  $\mu = k_i + km'$  with  $k \in \mathbb{Z}$ ; the integer  $k$  is null since  $0 < \mu < m'$  from the inequality (1); thus  $\mu' = 0$ )

This means that  $v_i'' = v_i$ ,  $v_{i+1}''' = v_{i+1}$  and the line segment  $[v_i, v_{i+1}]$  only cuts  $R$ -tiles. Thus  $v_i'$  and  $v_i$  both are on the left side of the  $R$ -tile  $t_1$  which yields that  $v_i' = v_i$  since their vertical coordinates are equal modulo  $n$ . With the same argument, we necessarily have  $v_{i+1}' = v_{i+1}$ . Thus we have

$$f_{2,T}(v_{i+1}) = f_{2,T}(v_i)(k_i m) \quad (3)$$

Now, consider the path starting in  $v_i$  by a horizontal move which follows the boundary of the polygon formed by the  $k_i$   $R$ -tiles of the sequence, until  $v_{i+1}$  is reached. This path directly induces a word, without possible simplification in  $G_2$  (since the (absolute) length of each vertical move is lower than  $n$ , and the length of each horizontal move is  $k_i m$  with  $k_i$  integer such that  $0 < k_i \leq k_i < m'$ ). This word is necessarily  $k_i m$ , from the equality 3. This means that no tile is cut by the segment  $[v_i, v_{i+1}]$ . This finishes the proof of the fact above, using induction.

Now, to get the result, it suffices to apply the fact for  $i = p$ . We necessarily have:  $v_p' = v_p$  since  $v_p$  finishes the path, thus the whole path cuts no tile.

### Application to Stairs of Blocks

**Definition 3.** A horizontal block (respectively vertical block) of  $F$  is a (closed)  $mm' \times n'$  rectangle (respectively  $m \times nn'$  rectangle).

A horizontal (respectively vertical) block  $B$  is covered by a block  $B'$  if the (open) top (respectively right) side of  $B$  and the (open) bottom (respectively left) side of  $B'$  share a line segment of length  $km$ , where  $k$  denotes a positive integer.

A stair of blocks (see figure 3) is a finite sequence  $(B_1, B_2, \dots, B_p)$  of blocks of the same type such that, for each integer  $i$  such that  $1 \leq i < p$ ,  $B_{i+1}$  covers  $B_i$ , and there exists no subsequence  $(B_{i+1}, B_{i+2}, \dots, B_{i+k})$  such that  $\cup_{j=1}^k B_{i+j}$  is an  $mm' \times nn'$  rectangle.

Notice that the figure formed by the union of blocks of a fixed stair has exactly one tiling which contains a unique type of tiles.

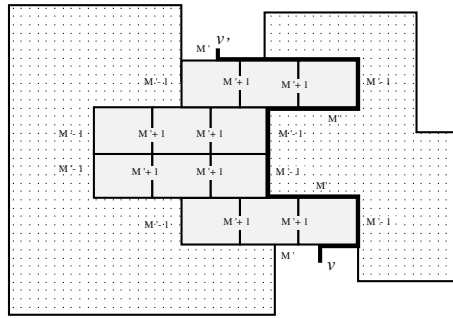
**Proposition 3.** Assume that there exists a stair  $(B_1, B_2, \dots, B_p)$  of horizontal blocks such that the intersection of  $B_1$  with the lower boundary of  $F$  contains a horizontal line segment  $S$  of length at least  $m$ , the intersection of  $B_p$  with the upper boundary of  $F$  contains a horizontal line segment  $S'$  of length at least  $m$ , and the tiles of the tiling of  $\cup_{i=1}^p B_i$  can be placed in  $F$  without inducing contradiction in the definition of the tiling function.

Each tiling  $T$  of  $F$  contains the tiling of  $\cup_{i=1}^p B_i$ .

*Proof.* From the hypothesis, there exists a vertex  $v$  of  $S$  such that the distance between  $v$  and the lower right corner of  $B_1$  is  $km$ , for an integer  $k$  such that  $0 < k < m'$ . In a similar way, there exists a vertex  $v'$  of  $S'$  such that the distance between  $v'$  and the upper right corner of  $B_p$  is  $k'm$ , for an integer  $k'$  such that  $0 < k' < m'$ .

The path from  $v$  to  $v'$  following the boundary of  $\cup_{i=1}^p B_i$  counterclockwise is a special path and, from the previous proposition, no tile of  $T$  cuts this path. The same argument can be used following the boundary of  $\cup_{i=1}^p B_i$  clockwise. This gives that no tile of  $T$  is cut by the boundary of  $\cup_{i=1}^p B_i$ .

Of course, we have a similar proposition for stairs of vertical blocks.



## 4 Height Function and Order

**Definition 4. (height, predecessor, order on  $G_1$ )** The height  $h_1(g)$  of an element  $g$  of  $G_1 = \mathbb{Z}/m\mathbb{Z} * \mathbb{Z}/n'\mathbb{Z}$  is defined by the following axioms :

- Given a pair  $(g, g')$  of elements of  $G_1$ , we say that  $g \leq g'$  if there exists a sequence  $(g_1, g_2, \dots, g_p)$  of elements of  $G_1$  such that  $g_1 = g$ ,  $g_p = g'$  and, for each integer  $i$  such that  $1 \leq i < p$ ,  $g_i$  is the predecessor of  $g_{i+1}$ .

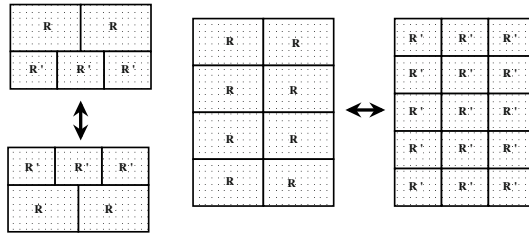
Notice that the heights of vertices which are on the same side  $[v', v'']$  of a tile form either a singleton  $\{h\}$  or a pair  $\{h, h-1\}$ . With these notations, we say that  $h$  is the height of  $[v', v'']$  in  $T$  (denoted by  $h_{1,T}([v', v''])$ ). With the convention:  $f_{1,T}(O) = 1_{G_1}$ , the height of horizontal sides is even, and the height of vertical sides is odd. Both vertical sides of an  $R$ -tile  $t$  have the same height, say  $h$  and the height of each horizontal tile is either  $h-1$  or  $h+1$  (a symmetric situation holds for  $R'$ -tiles). The height  $h_1(t)$  of the tile  $t$  is defined by:  $h_1(T) = h$  (i. e. the height of an  $R$ -tile (respectively  $R'$ -tile) is the height of its vertical (respectively horizontal) sides. The height  $h_1(T)$  of the tiling  $T$  is defined by:  $h_1(T) = \sum_{t \in T} a(t)h_{1,T}(t)$  where  $a(t)$  denotes the area of the tile  $t$ .



**Local Flips.** An  $mm' \times (n + n')$  rectangle admits two tilings. Let  $T$  be a tiling of  $F$  such that an  $mm' \times (n + n')$  rectangle is exactly covered by tiles of  $T$ . The replacement of those tiles of  $T$  (which cover the  $mm' \times (n + n')$  rectangle) by tiles of the other tiling of the same rectangle is called a *weak vertical flip* (see figure 4). By this way, a new tiling  $T_{flip}$  is obtained.

The flip is said *pushing up* if  $R'$ -tiles of  $T$  are in the bottom part of the  $mm' \times (n + n')$  rectangle (and  $R'$ -tiles of  $T_{flip}$  are in the top part of the  $mm' \times (n + n')$  rectangle). Otherwise, the flip is said *pushing down*. If the height of both horizontal sides of the  $mm' \times (n + n')$  rectangle differ by two units, then  $h_1(T)$  and  $h_1(T_{flip})$  are not equal. Otherwise  $h_1(T) = h_1(T_{flip})$  and the flip is said *height stable*.

In a symmetric way, we have another type of weak flips, the horizontal flips which based on the two tilings of an  $(m + m') \times nn'$  rectangles. We use some similar definitions (pushing left flip, pushing right flip, height stable flip).



**Fig. 4.** A weak flip on the left, a strong flip on the right

We have another type of flip (called *strong flip*) based on the fact that we have two ways to tile a  $mm' \times nn'$  rectangle (see figure 4). Such a flip always changes the value of  $h_1(T)$ .

**Definition 5. (equivalence, order on the set of tilings)** Two tilings are said height equivalent if one can pass from one to the other one by a sequence of height stable flips.

Given a pair  $(T, T')$  of tilings, we say that  $T$  covers  $T'$  is there exists a pair  $(T_{aux}, T'_{aux})$  of tilings of  $F$  such that  $T$  is equivalent to  $T_{aux}$ ,  $T'$  is equivalent to  $T'_{aux}$ ,  $h_1(T_{aux}) > h_1(T'_{aux})$ , and  $T_{aux}$  is deduced from  $T'_{aux}$  by one flip. The order relation which is the transitive closure of the relation of covering is denoted by  $>$ .

**Critical Tiling.** For the order defined above, there exists at least a minimal tiling. If a tiling is minimal, then all the height equivalent tilings are also minimal.

**Definition 6.** A critical tiling  $T_{crit}$ , is a minimal tiling such that neither height equivalent pushing down flip nor height equivalent pushing left flip can be done.

To prove that there exists a critical tiling, it suffices to consider consider the potential function  $pot$  defined as follows: for each tiling  $T$ ,  $pot(T)$  is the sum of the vertical coordinates of the horizontal sides of  $R'$ -tiles of  $T$  and the horizontal coordinates of

the vertical sides of  $R'$ -tiles. Notice that a pushing down (respectively left) flip removes  $2mn'$  (respectively  $2m'n$ ) units from the function  $pot$ . Thus, for any weak flip  $pot(T)$  and  $pot(T_{flip})$  are comparable. A minimal tiling which has the lowest potential value among the minimal tilings is necessarily critical.

**Proposition 4.** *Let  $T_{crit}$  be a critical tiling. Let  $M$  denote the highest value of  $h_{1,T_{crit}}$  in  $V_{T_{crit}}$  and  $M'$  denote the highest value of  $h_{1,T_{crit}}$  in  $\delta F$ .*

*We have :  $M' \geq M - 1$ . Moreover, when  $M' = M - 1$  and  $M$  is odd:*

- *there exists a set of blocks with pairwise disjoint interiors (called critical blocks) whose tilings are included in  $T_{crit}$ , for each vertex  $v$ ,  $h_{1,T_{crit}}(v) = M$  if and only if  $v$  is in the interior part of a line segment  $[v', v'']$  which is the common side of two  $R'$ -tiles of the same block.*
- *each critical block of  $T_{crit}$  whose bottom side does not share a line segment with  $\delta F$  covers another critical block of  $T_{crit}$*
- *there is no set of  $n$  critical blocks whose union is an  $mm' \times nn'$  rectangle,*
- *the value  $M'$  is reached on both lower and upper boundaries of  $F$ .*

We omit the proof. There exists a symmetric proposition for  $M$  even.

**Corollary 1.** *We keep the notations above. Assume that  $M'$  is even. For each vertex  $v$  of  $\delta F$  such that  $h_1(v) = M'$ , at least one of the following alternatives holds:*

- *the vertex  $v$  is on the boundary of an  $R$ -tile of  $T_{crit}$  with two vertical sides of height  $M' - 1$ ,*
- *the vertex  $v$  is on the boundary of the horizontal side of a critical block of  $T_{crit}$ . Moreover, if  $v$  is on the upper boundary of  $F$ , then there exists a stair  $(B_1, B_2, \dots, B_p)$  of critical blocks such that the block  $B_1$  meets the lower boundary of  $F$  and  $v$  is on the top side of  $B_p$ .*

*Proof.* Obvious, since any other situation contradicts Proposition 4.

**Algorithm of Tiling.** We present the algorithm below.

*Initialization:* Construct  $f_1, f_2$  on  $\delta F$  (if a contradiction holds, there is no tiling).

*Main loop:* Consider the set  $S_{M'}$  of vertices such that  $h_1(v) = M'$ . We assume that  $M'$  is even (the case when  $M'$  is odd is treated in a symmetric way). If  $S_{M'}$  is included in the lower boundary of  $F$ , then place the  $R$ -tile  $t$  in the neighborhood of a vertex  $v$  of  $S_{M'}$  in such a way that the maximal height reached in the boundary of  $t$  is  $M'$  (at most one possibility with no contradiction for  $f_{1,T}$  and  $f_{2,T}$ ). Otherwise, take a vertex  $v$  of  $S_{M'}$  in the upper boundary of  $F$ . If there exists a stair  $(B_1, B_2, \dots, B_p)$  of blocks such that the block  $B_1$  meets the lower boundary of  $F$ ,  $v$  is on the top side of  $B_p$ , and the stair creates no contradiction for  $f_{1,T}$  and  $f_{2,T}$ , then place the tiles of the tiling of the stair of blocks. If such a stair does not exist, then place the  $R$ -tile  $t$  in the neighborhood of the vertex  $v$  in such a way that the maximal height reached in the boundary of  $t$  is  $M'$  (at most one possibility with no contradiction for  $f_{1,T}$  and  $f_{2,T}$ ).

Remove the area tiled from  $F$ . Update  $M'$  and  $S_{M'}$ . Repeat the main loop until the remaining figure is empty or a contradiction appears about tiling functions.

**Corollary 2.** *Assume that  $F$  can be tiled. The above algorithm gives a critical tiling. In particular, there exists a unique critical tiling  $T_{crit}$ .*

*Proof.* From Proposition 4, if  $S_{M'}$  is included in the lower boundary of  $F$ , then  $M = M'$ , which guarantees the correctness of the placement of the tile in this alternative. On the other hand, Corollary 1 guarantees the correctness of the placement of tiles when  $S_{M'}$  contains a vertex of the upper boundary of  $F$ .

**Theorem 1. (connectivity)** *Let  $m, m', n$  and  $n'$  be fixed integers, each of them being larger than 2. Let  $(T, T')$  be a pair of tilings of a polygon  $F$  by translated copies of  $m \times n$  and  $m' \times n'$  rectangles. One can pass from  $T$  to  $T'$  by a sequence of local flips.*

*Proof.* Obvious, from Corollary 2.

**Theorem 2. (quadratic algorithm)** *Let  $m, m', n$  and  $n'$  be fixed integers, each of them being larger than 2. There exists a quadratic-time tiling algorithm (in the area  $A(F)$  of  $F$ ) which, given a polygon  $F$ , either produces a tiling of  $F$  by translated copies of  $m \times n$  and  $m' \times n'$  rectangles, or claims that there is no tiling (if it is the case).*

*Proof.* we have seen in Corollary 2 that the algorithm given above is a tiling algorithm. We just have to study its complexity. Each passage through the loop permits to place at least one tile, thus there are at most  $A(F)$  passages through the loop.

Each passage through the loop costs  $O(A(F))$  time units. The key-point is that the search of a stair of blocks crossing the polygon is similar to the depth first search in a tree which has  $O(A(F))$  vertices.

## 5 Related Problems and Extensions

**Case with a Side of Unit Length.** Until this section, it has been assumed that all the dimensions of tiles are at least 2. When a tile has a side of unit length, at least one of the tiling groups is degenerate, i. e. reduced to a cyclic group. In such a case, the same approach can be done, with some simplifications due to degeneration. We also have a connectivity result and the theorem below (which contains the results of [5] about bars):

**Theorem 3. (linear algorithm)** *Let  $m, m'$  and  $n'$  be fixed positive integers. There exists a linear-time tiling algorithm (in the area  $A(F)$  of  $F$ ) which, given a polygon  $F$ , either produces a tiling of  $F$  with translated copies of  $m \times 1$  and  $m' \times n'$  rectangles, or claims that there is no tiling (if it is the case).*

**Extensions.** The same ideas can be used for rectangles with real dimensions. One can prove a connectivity result and give an algorithm (there is no standard framework to study complexity when we have real numbers).

A motivating direction of research is a better knowledge of the structure of the space of tilings, i. e. the graph whose vertices are tilings of a polygon  $F$  and two tilings are linked by an edge if and only if they only differ by a single flip. Some very strong structural results (lattice structures, effective computations of geodesics between fixed tilings) have previously been found in the special case when each tile has a side of unit width [14]. Can we extend these results to the main case ?

## References

1. D. Beauquier, M. Nivat, E. Rémila, J. M. Robson, *Tiling figures of the plane with two bars*, Computational Geometry: Theory and Applications **5**, (1995) p. 1-25,
2. J. K. Burton, C. L. Henley, *A constrained Potts antiferromagnet model with an interface representation*, J. Phys. A **30** (1997) p. 8385-8413.
3. J. H. Conway, J. C. Lagarias, *Tiling with Polyominoes and Combinatorial Group Theory*, Journal of Combinatorial Theory A **53** (1990) p. 183-208.
4. C. Kenyon, personal communication (1992)
5. C. Kenyon, R. Kenyon, *Tiling a polygon with rectangles*, proceedings of the 33<sup>rd</sup> IEEE conference on Foundations of Computer Science (FOCS) (1992) p. 610-619.
6. R. Kenyon, *A note on tiling with integer-sided rectangles*, Journal of Combinatorial Theory A **74** (1996) p. 321-332.
7. J. C. Lagarias, D. S. Romano, *A Polyomino Tiling of Thurston and its Configurational Entropy*, Journal of Combinatorial Theory A **63** (1993) p. 338-358.
8. W. Magnus, A. Karrass, D. Solitar *Combinatorial Group Theory. Presentation of groups in terms of generators and relations*, 2<sup>nd</sup> edition (1976) Dover publications.
9. C. Moore, I. Rapaport, E. Rémila, *Tiling groups for Wang tiles*, proceedings of the 13<sup>th</sup> ACM-SIAM Symposium On Discrete Algorithms (SODA) SIAM eds, (2002) p. 402-411.
10. I. Pak, *Ribbon Tile Invariants*, Trans. Am. Math. Soc. **63** (2000) p. 5525-5561.
11. I. Pak, *Tile Invariants*, *New Horizons*, Theoretical Computer Science **303** (2003) p. 303-331.
12. J. G. Propp, *A Pedestrian Approach to a Method of Conway, or, A Tale of Two Cities*, Mathematics Magazine **70** (1997) p. 327-340.
13. E. Rémila, *Tiling groups : new applications in the triangular lattice*, Discrete and Computational Geometry **20** (1998) p. 189-204.
14. E. Rémila, *On the structure of some spaces of tilings*, SIAM Journal on Discrete Mathematics **16** (2002) p. 1-19
15. S. Sheffield, *Ribbon tilings and multidimensional height functions*, Transactions of the American Mathematical Society **354** (2002), p. 4789-4813.
16. W. P. Thurston, *Conway's tiling groups*, American Mathematical Monthly **97** (1990) p. 757-773.

# On Dynamic Shortest Paths Problems

Liam Roditty and Uri Zwick

School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel

**Abstract.** We obtain the following results related to dynamic versions of the shortest-paths problem:

- (i) Reductions that show that the incremental and decremental *single-source* shortest-paths problems, for weighted directed or undirected graphs, are, in a strong sense, at least as hard as the static *all-pairs* shortest-paths problem. We also obtain slightly weaker results for the corresponding unweighted problems.
- (ii) A randomized fully-dynamic algorithm for the all-pairs shortest-paths problem in directed unweighted graphs with an amortized update time of  $\tilde{O}(m\sqrt{n})$  and a worst case query time is  $O(n^{3/4})$ .
- (iii) A deterministic  $O(n^2 \log n)$  time algorithm for constructing a  $(\log n)$ -spanner with  $O(n)$  edges for any weighted undirected graph on  $n$  vertices. The algorithm uses a simple algorithm for incrementally maintaining single-source shortest-paths tree up to a given distance.

## 1 Introduction

The objective of a dynamic shortest path algorithm is to efficiently process an online sequence of update and query operations. Each update operation inserts or deletes edges from an underlying dynamic graph. Each query operation asks for the distance between two specified vertices in the current graph. A dynamic algorithm is said to be *fully dynamic* if it can handle both insertions and deletions. An *incremental* algorithm is an algorithm that can handle insertions, but not deletions, and a *decremental* algorithm is an algorithm that can handle deletions, but not insertions. Incremental and decremental algorithms are sometimes referred to as being *partially dynamic*. An *all-pairs* shortest paths (APSP) algorithm is an algorithm that can report distances between any two vertices of the graph. A *single-source* shortest paths (SSSP) algorithm can only report distances from a given source vertex.

We present three results related to dynamic shortest paths problems. We begin with simple reductions that show that the innocent looking incremental and decremental SSSP problems are, in a strong sense, at least as hard as the static APSP problem. This may explain the lack of progress on these problems, and indicates that it will be difficult to improve classical algorithms for these problems, such as the decremental algorithm of Even and Shiloach [9].

We then present a new fully dynamic APSP algorithm for unweighted directed graphs. The amortized update time of the algorithm is  $\tilde{O}(m\sqrt{n})$  and

the worst-case query time is  $O(n^{3/4})$ . The algorithm is randomized. The results returned by the algorithm are correct with very high probability. The new algorithm should be compared with a recent algorithm of Demetrescu and Italiano [8] and its slight improvement by Thorup [26]. Their algorithm, that works for *weighted* directed graphs, has an amortized update time of  $\tilde{O}(n^2)$  and a query time of  $O(1)$ . For sparse enough graphs our new algorithm has a faster update time. The query cost, alas, is much larger.

The new algorithm can also be compared to fully dynamic *reachability* algorithms for directed graphs obtained by the authors in [20] and [21]. A reachability algorithm is only required to determine, given two vertices  $u$  and  $v$ , whether there is a directed path from  $u$  to  $v$  in the graph. The reachability problem, also referred to as the *transitive closure* problem, is, of course, easier than the APSP problem. A fully dynamic reachability algorithm with an amortized update time of  $\tilde{O}(m\sqrt{n})$  and a worst-case query time of  $O(\sqrt{n})$  is presented in [20]. A fully dynamic reachability algorithm with an amortized update time of  $O(m+n \log n)$  and a worst-case query time of  $O(n)$  is presented in [21].

Finally, we present a simple application of incremental SSSP algorithms, showing that they can be used to speed up the operation of the greedy algorithm for constructing spanners. In particular, we obtain an  $O(n^2 \log n)$  time algorithm for constructing an  $O(\log n)$ -spanner with  $O(n)$  edges for any weighted undirected graph on  $n$  vertices. The previously fastest algorithm for constructing such spanners runs in  $O(mn)$  time.

The rest of this paper is organized as follows. In the next section we describe the hardness results for incremental and decremental SSSP. We also discuss the implications of these results. In Section 3 we then present our new fully dynamic APSP algorithm. In Section 4 we present our improved spanner construction algorithm. We end in Section 5 with some concluding remarks and open problems.

## 2 Hardness of Partially Dynamic SSSP Problems

We start with two simple reductions that show that the incremental and decremental *weighted* SSSP problems are at least as hard as the static weighted APSP problem. We then present two similar reductions that show that the incremental and decremental *unweighted* SSSP problems are at least as hard as several natural static graph problems such as Boolean matrix multiplication and the problem of finding all edges of a graph that are contained in triangles.

Let  $\mathcal{A}$  be an incremental (decremental) algorithm for the weighted (unweighted) directed (undirected) SSSP problem. We let  $init_{\mathcal{A}}(m, n)$  be the initialization time of  $\mathcal{A}$  on a graph with  $m$  edges and  $n$  vertices. We let  $update_{\mathcal{A}}(m, n)$  be the amortized edge insertion (deletion) time of  $\mathcal{A}$ , and  $query_{\mathcal{A}}(m, n)$  be the amortized query time of  $\mathcal{A}$ , where  $m$  and  $n$  are the number of edges and vertices in the graph at the time of the operation. We assume that the functions  $init_{\mathcal{A}}(m, n)$ ,  $update_{\mathcal{A}}(m, n)$  and  $query_{\mathcal{A}}(m, n)$  are monotone in  $m$  and  $n$ .

**Theorem 1.** *Let  $\mathcal{A}$  be an incremental (decremental) algorithm for the weighted directed (undirected) SSSP problem. Then, there is an algorithm for the static*

*APSP problem for weighted graphs that runs in  $O(\text{init}_{\mathcal{A}}(m+n, n+1) + n \cdot \text{update}_{\mathcal{A}}(m+n, n+1) + n^2 \cdot \text{query}_{\mathcal{A}}(m+n, n+1))$  time.*

*Proof.* Let  $G = (V, E)$  be a graph, with  $|V| = n$  and  $|E| = m$ , and let  $w : E \rightarrow \mathbb{R}^+$  be an assignment of non-negative weights to its edges. The proof works for both directed and undirected graphs. We assume, without loss of generality, that  $V = \{1, 2, \dots, n\}$ . Let  $W = \max_{e \in E} w(e)$  be the maximum edge weight.

Assume, at first, that  $\mathcal{A}$  is a decremental algorithm. We construct a new graph  $G_0 = (V \cup \{0\}, E \cup (\{0\} \times V))$ , where 0 is a new source vertex. A new edge  $(0, j)$ , where  $1 \leq j \leq n$  is assigned the weight  $j \cdot nW$ . (See Figure 1(a).) The graph  $G_0$ , composed of  $n+1$  vertices and  $m+n$  edges, is passed as the initial graph to the decremental algorithm  $\mathcal{A}$ . The source is naturally set to be 0. After  $\mathcal{A}$  is initialized, we perform the  $n$  queries  $\text{query}(j)$ , for  $1 \leq j \leq n$ . Each query  $\text{query}(j)$  returns  $\delta_{G_0}(0, j)$ , the distance from 0 to  $j$  in  $G_0$ . As the weight of the edge  $(0, 1)$  is substantially smaller than the weight of all other edges emanating from the source, it is easy to see that  $\delta_G(1, j) = \delta_{G_0}(0, j) - nW$ , for every  $1 \leq j \leq n$ . We now delete the edge  $(0, 1)$  from  $G_0$  and perform again the  $n$  queries  $\text{query}(j)$ , for  $1 \leq j \leq n$ . We now have  $\delta_G(2, j) = \delta_{G_0}(0, j) - 2nW$ , for every  $1 \leq j \leq n$ . Repeating this process  $n-2$  more times we obtain all distances in the original graph by performing only  $n$  edge deletions and  $n^2$  queries.

The proof when  $\mathcal{A}$  is an incremental algorithm is analogous. The only difference is that we now insert the edges  $(0, j)$  one by one, in reverse order. We first insert the edge  $(0, n)$ , with weight  $n^2W$ , then the edge  $(0, n-1)$  with weight  $(n-1)nW$ , and so on.  $\square$

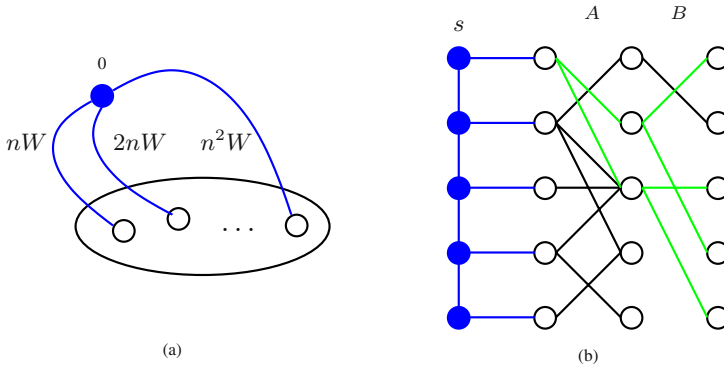
We note that the simple reduction just described works for undirected, directed, as well as acyclic directed graphs (DAGs). We next move to unweighted versions of the problem.

**Theorem 2.** *Let  $\mathcal{A}$  be an incremental (decremental) algorithm for the unweighted directed (undirected) SSSP problem. Then, there is an algorithm that multiplies two Boolean  $n \times n$  matrices, with a total number of  $m$  1's, in  $O(\text{init}_{\mathcal{A}}(m+2n, 4n) + n \cdot \text{update}_{\mathcal{A}}(m+2n, 4n) + n^2 \cdot \text{query}_{\mathcal{A}}(m+2n, 4n))$  time.*

*Proof.* Let  $A$  and  $B$  be two Boolean  $n \times n$  matrices. Let  $C = AB$  be their Boolean product. Construct a graph  $G = (V, E)$  as follows:  $V = \{s_i, u_i, v_i, w_i \mid 1 \leq i \leq n\}$ , and  $E = \{(s_i, s_{i+1}) \mid 1 \leq i < n\} \cup \{(s_i, u_i) \mid 1 \leq i \leq n\} \cup \{(u_i, v_j) \mid a_{ij} = 1, 1 \leq i, j \leq n\} \cup \{(v_i, w_j) \mid b_{ij} = 1, 1 \leq i, j \leq n\}$ . (See Figure 1(b).) The graph  $G$  is composed of  $4n$  vertices and  $m+2n-1$  edges. Let  $s = s_1$ . It is easy to see that  $\delta_G(s, w_j) = 3$  if and only if  $c_{1j} = 1$ . We now delete the edge  $(s_1, u_1)$ . Now,  $\delta_G(s, w_j) = 4$  if and only if  $c_{2j} = 1$ . We then delete the edge  $(s_2, u_2)$ , and so on. Again we use only  $n$  delete operations and  $n^2$  queries. The incremental case is handled in a similar manner.  $\square$

**Discussion.** All known algorithms for the static APSP problems in weighted directed or undirected graphs run in  $\Omega(mn)$  time. A running time of  $O(mn + n^2 \log n)$  is obtained by running Dijkstra's algorithm from each vertex (see [10]). Slightly faster algorithms are available, in various settings. For the best available





**Fig. 1.** Reductions of static problems to incremental or decremental SSSP problems

results see [10], [25], [13], [18], [17]. Karger *et al.* [15] show that any *path-comparison* algorithm for the problem must have a running time of  $\Omega(mn)$ .

The reduction of Theorem 1 shows that if there is an incremental or decremental SSSP algorithm that can handle  $n$  update operations and  $n^2$  query operations in  $o(mn)$  time, then there is also an  $o(mn)$  time algorithm for the static APSP problem. We note that the trivial ‘dynamic’ SSSP algorithm that simply constructs a shortest paths tree from scratch after each update operation handles  $n$  update operations in  $\tilde{O}(mn)$  time. Almost any improvement of this trivial algorithm, even with much increased query times, will yield improved results for the static APSP problem.

An interesting open problem is whether there are incremental or decremental SSSP algorithms for *weighted* graphs that can handle  $m$  updates and  $n^2$  queries in  $\tilde{O}(mn)$  time. (Note that the number of updates here is  $m$  and not  $n$ .)

We next consider *unweighted* versions of the SSSP problem. A classical result in this area is the following:

**Theorem 3 (Even and Shiloach [9]).** *There is a decremental algorithm for maintaining the first  $k$  levels of a single-source shortest-paths tree, in a directed or undirected unweighted graph, whose total running time, over all deletions, is  $O(km)$ , where  $m$  is the initial number of edges in the graph. Each query can be answered in  $O(1)$  time.*

It is easy to obtain an incremental variant of this algorithm. Such a variant is described, for completeness, in Section 4, where it is also used.

How efficient is the algorithm of [9], and what are the prospects of improving it? If  $k$ , the number of levels required is small, then the running time of the algorithm is close to be optimal, as  $\Omega(m)$  is an obvious lower bound. But, if a complete shortest paths tree is to be maintained, i.e.,  $k = n - 1$ , the running time of the algorithm becomes  $O(mn)$ . How hard will it be to improve this result?

Our reductions for the unweighted problems are slightly weaker than the ones we have for the weighted problems. We cannot reduce the static APSP problems to the partially dynamic SSSP problems, but we can still reduce the Boolean



matrix multiplication problem to them. The APSP problem for undirected unweighted graphs can be reduced to the Boolean matrix multiplication problem (see [11],[23],[24]), but these reductions does not preserve sparsity.

The fastest known combinatorial algorithm for computing the Boolean product of two  $n \times n$  matrices that contain a total of  $m$  1's runs in  $O(mn)$  time. By a combinatorial algorithm here we refer to an algorithm that does not rely on fast algebraic matrix multiplication techniques. Using such algebraic techniques it is possible to multiply the matrices in  $O(n^{2.38})$  time (see [7]), and also in  $O(m^{0.7}n^{1.2} + n^2)$  time (see [29]). Obtaining a combinatorial Boolean matrix multiplication algorithm whose running time is  $O((mn)^{1-\epsilon} + n^2)$ , or  $O(n^{3-\epsilon})$ , for some  $\epsilon > 0$ , is a major open problem.

The reduction of Theorem 2 shows that reducing the total running time of the algorithm of [9] to  $o(mn)$ , using only combinatorial means, is at least as hard as obtaining an improved combinatorial Boolean matrix multiplication algorithm. Also, via the reduction of the static APSP problem to Boolean matrix multiplication, we get that an incremental or decremental SSSP algorithm with a total running time of  $O(n^{3-\epsilon})$ , and a query time of  $O(n^{1-\epsilon})$ , for some  $\epsilon > 0$ , will yield a combinatorial algorithm for the static APSP problem with a running time of  $O(n^{3-\epsilon})$ . We believe that this provides strong evidence that improving the algorithm of [9] will be very hard.

It is also not difficult to see that if the first  $k$  levels of a single-source shortest-paths tree can be incrementally or decrementally maintained in  $o(km)$  time, then there is an  $o(mn)$  time Boolean matrix multiplication algorithm. The details will appear in the full version of the paper.

Chan [5] describes a simple reduction from the *rectangular* Boolean matrix multiplication problem to the *fully dynamic subgraph connectivity* problem. It is similar in spirit to our reduction. The details, and the problems involved, are different, however.

As a final remark we note that we have reduced the APSP problem and the Boolean matrix multiplication problem to *offline* versions of incremental or decremental SSSP problem. It will thus be difficult to obtain improved algorithms for partially dynamic SSSP problems even if all the update and query operations are given in advance.

### 3 Fully Dynamic All-Pairs Shortest Paths

In this section we obtain a new fully dynamic algorithm for the all-pairs shortest paths problem. The algorithm relies on ideas of [14] and [20]. We rely on following result of [14] and a simple observation of [28]:

**Theorem 4 (Henzinger and King [14]).** *There is a randomized decremental all-pairs shortest-paths algorithm for directed unweighted graphs whose total running time, over all deletions, is  $O(\frac{mn^2 \log n}{t} + mn \log^2 n)$  and whose query time is  $O(t)$ , where  $m$  and  $n$  are the number of edges and vertices in the initial graph, and  $t \geq 1$  is a parameter. (In particular, for  $t \leq n/\log n$ , the total run-*

ning time is  $O(\frac{mn^2 \log n}{t})$ .) Every result returned by the algorithm is correct with a probability of at least  $1 - n^{-c}$ , where  $c$  is a parameter set in advance.

**Lemma 1 (Ullman and Yannakakis [28]).** *Let  $G = (V, E)$  be a directed graph on  $n$  vertices. Let  $1 \leq k \leq n$ , and let  $S$  be a random subset of  $V$  obtained by selecting each vertex, independently, with probability  $p = (c \ln n)/k$ . (If  $p \geq 1$ , we let  $S$  be  $V$ .) If  $p$  is a path in  $G$  of length at least  $k$ , then with a probability of at least  $1 - n^{-c}$ , at least one of the vertices on  $p$  belongs to  $S$ .*

The new algorithm works in *phases* as follows. In the beginning of each phase, the current graph  $G = (V, E)$  is passed to the decremental algorithm of [14] (Theorem 4). A random subset  $S$  of the vertices, of size  $(cn \ln n)/k$ , is chosen, where  $k$  is a parameter to be chosen later. The standard BFS algorithm is then used to build shortest paths trees to and from all the vertices of  $S$ . If  $w \in V$ , we let  $T_{in}(w)$  be a tree of shortest paths *to*  $w$ , and  $T_{out}(w)$  be a tree of shortest paths *from*  $w$ . The set  $C$  is initially empty.

An insertion of a set  $E'$  of edges, all touching a vertex  $v \in V$ , said to be the *center* of the insertion, is handled as follows. First if  $|C| \geq t$ , where  $t$  is a second parameter to be chosen later, then the current phase is declared over, and all the data structures are reinitialized. Next, the center  $v$  is added to the set  $C$ , and the first  $k$  levels of shortest paths trees  $\hat{T}_{in}(v)$  and  $\hat{T}_{out}(v)$ , containing shortest paths to and from  $v$ , are constructed. The trees  $\hat{T}_{in}(v)$  and  $\hat{T}_{out}(v)$  are constructed and maintained using the algorithm of [9] (Theorem 3). Finally, shortest paths trees  $T_{in}(w)$  and  $T_{out}(w)$ , for every  $w \in S$ , are constructed from scratch. (Note that we use  $\hat{T}_{in}(v)$  and  $\hat{T}_{out}(v)$  to denote the trees associated with a vertex  $v \in C$ , and  $T_{in}(w)$  and  $T_{out}(w)$ , without the hats, to denote the trees of a vertex  $w \in S$ . The former are decrementally maintained, up to depth  $k$ , while the later are rebuilt from scratch following each update operation.)

A deletion of an arbitrary set  $E'$  of edges is handled as follows. First, the edges of  $E'$  are removed from the decremental data structure initialized at the beginning of the current phase, using the algorithm of [14] (Theorem 4). Next, the algorithm of [9] (Theorem 3) is used to update the shortest paths trees  $\hat{T}_{in}(v)$  and  $\hat{T}_{out}(v)$ , for every  $v \in C$ . Finally, the trees  $T_{in}(w)$  and  $T_{out}(w)$ , for every  $w \in S$ , are again rebuilt from scratch.

A distance query  $Query(u, v)$ , asking for the distance  $d(u, v)$  from  $u$  to  $v$  in the current version of the graph, is handled using the following three stage process. First, we query the decremental data structure, that keeps track of all delete operations performed in the current phase, but ignores all insert operations, and get an answer  $\ell_1$ . We clearly have  $d(u, v) \leq \ell_1$ , as all edges in the decrementally maintained graph are also edges of the current graph. Furthermore, if there is a shortest path from  $u$  to  $v$  in the current graph that does not use any edge that was inserted during the current phase, then  $d(u, v) = \ell_1$ .

Next, we try to find a shortest path from  $u$  to  $v$  that passes through one of the insertion centers contained in  $C$ . For every  $w \in C$ , we query  $\hat{T}_{in}(w)$  for the distance from  $u$  to  $w$  and  $\hat{T}_{out}(w)$  for the distance from  $w$  to  $v$ , and add these two numbers. (If  $d(u, w) > k$ , then  $u$  is not contained in  $\hat{T}_{in}(w)$  and the distance from  $w$  to  $u$ , in the present context, is taken to be  $\infty$ . The case  $d(w, v) > k$

**Init( $G, k, t$ ):**

1.  $Init-Dec(G, t)$
2.  $C \leftarrow \phi$
3.  $S \leftarrow Random(V, (cn \ln n)/k)$
4.  $Build-Trees(S)$

**Delete( $E'$ ):**

1.  $E \leftarrow E - E'$
2.  $Delete-Dec(E')$
3. for every  $v \in C$
4.    $Delete-Tree(\hat{T}_{in}(v), E', k)$
5.    $Delete-Tree(\hat{T}_{out}(v), E', k)$
6.  $Build-Trees(S)$

**Insert( $E', v$ ):**

1.  $E \leftarrow E \cup E'$
2. if  $|C| \geq t$  then  $Init(G, k, t)$
3.  $C \leftarrow C \cup \{v\}$
4.  $Init-Tree(\hat{T}_{in}(v), E, k)$
5.  $Init-Tree(\hat{T}_{out}(v), E, k)$
6.  $Build-Trees(S)$

**Build-Trees( $S$ ):**

1. for every  $w \in S$
2.    $BFS(T_{in}(w), E)$
3.    $BFS(T_{out}(w), E)$

**Query( $u, v$ ):**

1.  $\ell_1 \leftarrow Query-Dec(u, v)$
2.  $\ell_2 \leftarrow \min_{w \in C} Query-Tree(\hat{T}_{in}(w), u) + Query-Tree(\hat{T}_{out}(w), v)$
3.  $\ell_3 \leftarrow \min_{w \in S} Query-Tree(T_{in}(w), u) + Query-Tree(T_{out}(w), v)$
4. return  $\min\{\ell_1, \ell_2, \ell_3\}$

**Fig. 2.** The new fully dynamic all-pairs shortest paths algorithm.

is handled similarly.) By taking the minimum of all these numbers we get a second answer that we denote by  $\ell_2$ . Again, we have  $d(u, v) \leq \ell_2$ . Furthermore, if  $d(u, v) \leq k$ , and there is a shortest path from  $u$  to  $v$  in the current graph that passes through a vertex that was an insertion center in the current phase of the algorithm, then  $d(u, v) = \ell_2$ .

Finally, we look for a shortest path from  $u$  to  $v$  that passes through a vertex of  $S$ . This is done in a similar manner by examining the trees associated with the vertices of  $S$ . The answer obtained using this process is denoted by  $\ell_3$ . (If there is no path from  $u$  to  $v$  that passes through a vertex of  $S$ , then  $\ell_3 = \infty$ .) The final answer returned by the algorithm is  $\min\{\ell_1, \ell_2, \ell_3\}$ .

A formal description of the new algorithm is given in Figure 2. The algorithm is initialized by a call  $Init(G, k, t)$ , where  $G = (V, E)$  is the initial graph and  $k$  and  $t$  are parameters to be chosen later. Such a call is also made at the beginning of each phase. A set  $E'$  of edges, centered at  $v$ , is added to the graph by a call  $Insert(E', v)$ . A set  $E'$  of edges is deleted by a call  $Delete(E')$ . A query is answered by calling  $Query(u, v)$ . A call  $Build-Trees(S)$  is used to (re)build shortest paths trees to and from the vertices of  $S$ .

The call  $Init-Dec(G, t)$ , in line 1 of  $Init$ , initializes the decremental algorithm of [14]. The call  $Random(V, (cn \ln n)/k)$ , in line 3, chooses the random sample  $S$ . The call  $Build-Trees(S)$ , in line 4, construct the shortest paths trees  $T_{in}(w)$  and  $T_{out}(w)$ , for every  $w \in S$ . A call  $Init-Tree(\hat{T}_{in}(v), E, k)$  (line 4 of  $Insert$ ) is used to initialize the decremental maintenance of the first  $k$  levels of a shortest paths tree

$\hat{T}_{in}(v)$  to  $v$ . Such a tree is updated, following a deletion of a set  $E'$  of edges, using a call *Delete-Tree*( $\hat{T}_{in}(v), E'$ ) (line 4 of *Delete*). A query *Query-Tree*( $\hat{T}_{in}(w), u$ ) (line 2 of *Query*) is used to find the distance from  $u$  to  $w$  in the tree  $\hat{T}_{in}(w)$ . If  $u$  is not in  $\hat{T}_{in}(w)$ , the value returned is  $\infty$ . Such a tree-distance query is easily handled in  $O(1)$  time. The out-trees  $\hat{T}_{out}(v)$  are handled similarly. Finally a call *BFS*( $T_{in}(w), E$ ) (line 2 of *Build-Trees*) is used to construct a standard, static, shortest paths tree to  $w$ . Distances in such trees are again found by calling *Query-Tree*( $T_{in}(w), u$ ) (line 3 of *Query*).

**Theorem 5.** *The fully dynamic all-pairs shortest paths algorithm of Figure 2 handles each insert or delete operation in  $O(\frac{mn^2 \log n}{t^2} + km + \frac{mn \log n}{k})$  amortized time, and answers each distance query in  $O(t + \frac{n \log n}{k})$  worst-case time. Each result returned by the algorithm is correct with a probability of at least  $1 - 2n^{-c}$ . By choosing  $k = (n \log n)^{1/2}$  and  $(n \log n)^{1/2} \leq t \leq n^{3/4}(\log n)^{1/4}$  we get an amortized update time of  $O(\frac{mn^2 \log n}{t^2})$  and a worst-case query time of  $O(t)$ .*

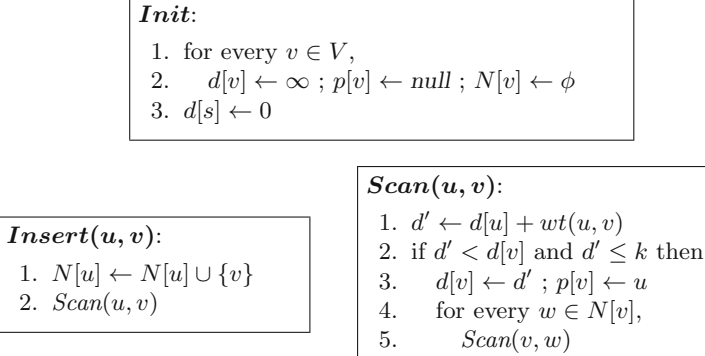
*Proof.* The correctness proof follows from the arguments outlined along side the description of the algorithm. As each estimate  $\ell_1, \ell_2$  and  $\ell_3$  obtained while answering a distance query *Query*( $u, v$ ) is equal to the length of a path in the graph from  $u$  to  $v$ , we have  $d(u, v) \leq \ell_1, \ell_2, \ell_3$ . We show that at least one of these estimates is equal, with very high probability, to  $d(u, v)$ .

If there is a shortest path from  $u$  to  $v$  that does not use any edge inserted in the current phase, then  $d(u, v) = \ell_1$ , assuming that the estimate  $\ell_1$  returned by the decremental data structure is correct. The error probability here is only  $n^{-c}$ .

Suppose therefore that there is a shortest path  $p$  from  $u$  to  $v$  that uses at least one edge that was inserted during the current phase. Let  $w$  be the *latest* vertex on  $p$  to serve as an insertion center. If  $d(u, v) \leq k$ , then the correct distance from  $u$  to  $v$  will be found while examining the trees  $\hat{T}_{in}(w)$  and  $\hat{T}_{out}(w)$ .

Finally, suppose that  $d(u, v) \geq k$ . Let  $p$  be a shortest path from  $u$  to  $v$  in the current graph. By Lemma 1, with a probability of at least  $1 - n^{-c}$  the path  $p$  passes through a vertex  $w$  of  $S$ , and the correct distance will be found while examining the trees  $T_{in}(w)$  and  $T_{out}(w)$ .

We next analyze the complexity of the algorithm. By Theorem 4, the total cost of maintaining the decremental data structure is  $O(\frac{mn \log^2 n}{t})$ . As each phase is composed of at least  $t$  update operations, this contributes  $O(\frac{mn \log^2 n}{t^2})$  to the amortized cost of each update operation. Each insert operation triggers the creation (or recreation) of two decremental shortest paths trees that are maintained only up to depth  $k$ . By Theorem 3 the total cost of maintaining these trees is only  $O(km)$ . (Note that this also covers the cost of all future operations performed on these trees.) Finally, each insert or delete operation requires the rebuilding of  $(cn \ln n)/k$  shortest paths trees at a total cost of  $O(\frac{mn \log n}{k})$ . The total amortized cost of each update operation is therefore  $O(\frac{mn^2 \log n}{t^2} + km + \frac{mn \log n}{k})$ , as claimed. Each query is handled by the algorithm in  $O(t + \frac{n \log n}{k})$ : The estimate  $\ell_1$  is obtained in  $O(t)$  time by querying the decremental data structure. The estimate  $\ell_2$  is obtained in  $O(t)$  by considering



**Fig. 3.** A simple incremental SSSP algorithm.

all the trees associated with  $C$ . Finally the estimate  $\ell_3$  is obtained in  $O(\frac{n \log n}{k})$  time by examining all the trees associated with  $S$ .

By examining these bounds it is obvious that  $k = (n \log n)^{1/2}$  is the optimal choice for  $k$ . By choosing  $t$  in the range  $(n \log n)^{1/2} \leq t \leq n^{3/4}(\log n)^{1/4}$ , we get a tradeoff between the update and query times. The fastest update time of  $O(m(n \log n)^{1/2})$  is obtained by choosing  $t = n^{3/4}(\log n)^{1/4}$ .  $\square$

## 4 An Incremental SSSP Algorithm and Greedy Spanners

A simple algorithm for incrementally maintaining a single-source shortest-paths tree from a source vertex  $s$  up to distance  $k$  is given in Figure 3. The edge weights are assumed to be non-negative integers. The algorithm may be seen as an incremental variant of the algorithm of [9]. It is also similar to an algorithm of Ramalingam and Reps [19]. The algorithm is brought here for completeness.

For each vertex  $v \in V$ ,  $d[v]$  is the current distance from  $s$  to  $v$ ,  $p[v]$  is the parent of  $v$  in the shortest paths tree, and  $N[v]$  are the vertices that can be reached from  $v$  by following an outgoing edge. The integer weight of an edge  $(u, v)$  is denoted by  $wt(u, v)$ . As described, the algorithm works on directed graphs. It is easy to adapt it to work on undirected graphs. (We simply need to scan each edge in both directions.)

**Theorem 6.** *The algorithm of Figure 3 incrementally maintains a shortest-paths tree from a source vertex  $s$  up to distance  $k$  in a directed unweighted graph using a total number of  $O(km)$  operations, where  $m$  is the number of edges in the final graph. Each distance query is answered in  $O(1)$  time.*

*Proof.* (Sketch) It is easy to see that the algorithm correctly maintains the distances from  $s$ . The complexity is  $O(km)$  as each edge  $(u, v)$  is rescanned only when the distance from  $s$  to  $u$  decreases, and this happens at most  $k$  times.  $\square$

We next define the notion of *spanners*.

***Greedy-Spanner*( $G, k$ ):**

1.  $E' \leftarrow \emptyset$
2. **for each** edge  $(u, v) \in E$ , in non-decreasing order of weight, **do**
3.   **if**  $\delta_{E'}(u, v) > (2k - 1) \cdot wt(u, v)$  **then**
- 3'.   **[if**  $d_{E'}(u, v) > (2k - 1)$  **then]**
4.        $E' \leftarrow E' \cup \{(u, v)\}$
5. **return**  $G' \leftarrow (V, E')$

**Fig. 4.** A greedy algorithm for constructing spanners.

**Definition 1 (Spanners [16]).** Let  $G = (V, E)$  be a weighted undirected graph, and let  $t \geq 1$ . A subgraph  $G' = (V, E')$  is said to be a  $t$ -spanner of  $G$  if and only if for every  $u, v \in V$  we have  $\delta_{G'}(u, v) \leq t \cdot \delta_G(u, v)$ .

The greedy algorithm of Althöfer et al. [1] for constructing sparse spanners of weighted undirected graphs is given in Figure 4. For every integer  $k \geq 2$ , it constructs a  $(2k - 1)$ -spanner with at most  $n^{1+1/k}$  edges. This is an essentially optimal tradeoff between stretch and size. The algorithm is reminiscent of Kruskal's algorithm for the construction of a minimum spanning tree algorithm. A naive implementation of this algorithm requires  $O(mn^{1+1/k})$  time.

We consider a variant of the algorithm in which line 3 is replaced by line 3'. For every edge  $(u, v) \in E$ , the original algorithm checks whether  $\delta_{E'}(u, v) > (2k - 1)wt(u, v)$ , i.e., whether the *weighted* distance from  $u$  to  $v$  in the subgraph composed of the edges already selected to the spanner is at most  $2k - 1$  times the weight  $wt(u, v)$  of the edge. The modified version of the algorithm asks, instead, whether  $d_{E'}(u, v) > 2k - 1$ , i.e., whether the *unweighted* distance between  $u$  and  $v$  in the subgraph  $(V, E')$  is greater than  $2k - 1$ . We now claim:

**Theorem 7.** *The modified version of the greedy spanner algorithm still produces a  $(2k - 1)$ -spanner with at most  $n^{1+1/k}$  edges for any weighted graph on  $n$  vertices.*

*Proof.* The claim follows from a simple modification of the correctness proof of the greedy algorithm. If an edge  $(u, v)$  is not selected by the modified algorithm, then  $d_{E'}(u, v) \leq 2k - 1$ . As the edges are scanned in an increasing order of weight, all the edges on the shortest path connecting  $u$  and  $v$  in  $(V, E')$  are of weight at most  $wt(u, v)$ , and therefore  $\delta_{E'}(u, v) \leq (2k - 1) \cdot wt(u, v)$ . Thus, the edge  $(u, v)$  is also not selected by the original algorithm. The edge set returned by the modified algorithm is therefore a superset of the edge set returned by the original algorithm, and is therefore a  $(2k - 1)$ -spanner of  $G$ .

The proof that the set of edges  $E'$  returned by the original algorithm is of size at most  $n^{1+1/k}$  relies only on the fact that the *girth* of  $G' = (V, E')$  is at least  $2k + 1$ . This also holds for the set  $E'$  constructed by the modified algorithm, as we never add to  $E'$  an edge that would form a cycle of size at most  $2k$ . Hence, the size of the set  $E'$  returned by the modified algorithm is also at most  $n^{1+1/k}$ .  $\square$

**Theorem 8.** *The modified greedy algorithm of Figure 4 can be implemented to run in  $O(kn^{2+1/k})$  time.*

*Proof.* We use the algorithm of Figure 3 to maintain a tree of shortest-paths, up to distance  $2k - 1$ , from each vertex of the graph. As the spanner contains at most  $n^{1+1/k}$  edges, the total cost of maintaining each one of these trees is only  $O(kn^{1+1/k})$ , and the total cost of the algorithm is  $O(kn^{2+1/k})$ , as claimed.  $\square$

**Discussion.** There are several other algorithms for constructing sparse spanners of weighted graphs. In particular, a randomized algorithm of Baswana and Sen [4] constructs a  $(2k - 1)$ -spanner with  $O(kn^{1+1/k})$  edges in  $O(m)$  expected time. A randomized  $O(mn^{1/k})$  algorithm for constructing such spanners is described in [27]. Why then insist on a faster implementation of the greedy algorithm? The answer is that the greedy algorithm constructs slightly sparser spanners. It produces  $(2k - 1)$ -spanners with at most  $n^{1+1/k}$  edges (no big- $O$  is needed here). When  $k$  is non-constant, this is significant. When we let  $k = \log n$ , the greedy algorithm produces an  $O(\log n)$ -spanner containing only  $O(n)$  edges. All other algorithms produce spanners with  $\Omega(n \log n)$  edges. It is, of course, an interesting open problem whether such spanners can be constructed even faster.

Another interesting property of the (original) greedy algorithm, shown by [6], is that the total weight of the edges in the  $(2k - 1)$ -spanner that it constructs is at most  $O(n^{(1+\epsilon)/k} \cdot wt(MST(G)))$ , for any  $\epsilon > 0$ , where  $wt(MST(G))$  is the weight of the minimum spanning tree of  $G$ . Unfortunately, this property no longer holds for the modified greedy algorithm. Again, it is an interesting open problem to obtain an efficient spanner construction algorithm that does have this property.

An efficient implementation of a different variant of the greedy algorithm, in the setting of geometric graphs, is described in [12].

## 5 Concluding Remarks and Open Problems

We presented a simple reduction from the static APSP problem for weighted graphs to offline partially dynamic SSSP problem for weighted graphs, and a simple reduction from the Boolean matrix multiplication problem to the offline partially dynamic SSSP problem for unweighted graphs.

An interesting issue to explore is whether faster partially dynamic SSSP algorithms may be obtained if *approximate* answers are allowed. (For steps in this direction, but for the approximate dynamic APSP problem, see [2,3,22].)

## References

1. I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9:81–100, 1993.
2. S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for transitive closure and all-pairs shortest paths. In *Proc. of 34th STOC*, pages 117–123, 2002.
3. S. Baswana, R. Hariharan, and S. Sen. Maintaining all-pairs approximate shortest paths under deletion of edges. In *Proc. of 14th SODA*, pages 394–403, 2003.
4. S. Baswana and S. Sen. A simple linear time algorithm for computing  $(2k - 1)$ -spanner of  $O(n^{1+1/k})$  size for weighted graphs. In *Proc. of 30th ICALP*, pages 384–296, 2003.



5. T. Chan. Dynamic subgraph connectivity with geometric applications. In *Proc. of 34th STOC*, pages 7–13, 2002.
6. B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. *Internat. J. Comput. Geom. Appl.*, 5:125–144, 1995.
7. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.
8. C. Demetrescu and G. Italiano. A new approach to dynamic all pairs shortest paths. In *Proc. of 35th STOC*, pages 159–166, 2003.
9. S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, 1981.
10. M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.
11. Z. Galil and O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134:103–139, 1997.
12. J. Gudmundsson, C. Levkopoulos, and G. Narasimhan. Fast greedy algorithm for constructing sparse geometric spanners. *SIAM J. Comput.*, 31:1479–1500, 2002.
13. T. Hagerup. Improved shortest paths on the word RAM. In *Proc. of 27th ICALP*, pages 61–72, 2000.
14. M. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proc. of 36th FOCS*, pages 664–672, 1995.
15. D. Karger, D. Koller, and S. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22:1199–1217, 1993.
16. D. Peleg and A. Schäffer. Graph spanners. *J. Graph Theory*, 13:99–116, 1989.
17. S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004.
18. S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proc. of 13th SODA*, pages 267–276, 2002.
19. G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, 1996.
20. L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proc. of 43rd FOCS*, pages 679–688, 2002.
21. L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proc. of 36th STOC*, pages 184–191, 2004.
22. L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. In *Proc. of 45th FOCS*, 2004. To appear.
23. R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51:400–403, 1995.
24. A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. of 40th FOCS*, pages 605–614, 1999.
25. M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.
26. M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Proc. of 9th SWAT*, 2004. To appear.
27. M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. of 33rd STOC*, pages 183–192, 2001. Full version to appear in the *Journal of the ACM*.
28. J. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20:100–125, 1991.
29. R. Yuster and U. Zwick. Fast sparse matrix multiplication. In *Proc. of 12th ESA*, 2004.



# Uniform Algorithms for Deterministic Construction of Efficient Dictionaries

Milan Ružić

The Faculty of Mathematics, Belgrade, Serbia and Montenegro  
mruzic@eunet.yu

**Abstract.** We consider the problem of linear space deterministic dictionaries over the universe  $\{0, 1\}^w$ . The model of computation is the unit-cost RAM with word length  $w$ . Many modern solutions to the dictionary problem are weakly non-uniform, i.e. they require a number of constants to be computed at “compile time” for stated time bounds to hold. We present an improvement in the class of dictionaries free from weak non-uniformity and with stress on faster searches. Various search-update time trade-offs are obtained. The time bounds for the construction of a static dictionary and update bounds for dynamic case contain  $\log w$  term; the query times are independent of  $w$ . In the case of optimal query time, we achieve construction time  $O(n^{1+\epsilon} \log w)$ , for any  $\epsilon > 0$ . The construction requires division, whereas searching uses multiplication only.

## 1 Introduction

A dictionary is a data structure that stores a set of keys  $S$  and supports answering of membership queries “Is  $x$  in  $S$ ?”. Keys come from the universe  $U = \{0, 1\}^w$  and may be accompanied by *satellite data* which can be retrieved in case  $x \in S$ . A dictionary is called *dynamic* if it also supports updates of  $S$  through insertions and deletions of elements. Otherwise, the dictionary is called *static*.

There are several characteristics that make distinction between various types of dictionaries, most important being: the amount of space occupied by a dictionary, the time needed for performing a query and the time taken by an update or a construction of the whole dictionary. Here, we consider only dictionaries with realistic space usage of  $O(n)$  registers, with  $n = |S|$ . Search time is given the priority over update times.

Our model of computation is the *word RAM* whose instruction set includes multiplication and division; all instructions can be completed in one unit of time. Elements of  $U$  are supposed to fit in one machine word and may be interpreted either as integers from  $\{0, \dots, 2^w - 1\}$  or as bit strings from  $\{0, 1\}^w$ .

Algorithms involved in construction of a dictionary may be randomized – they require a source of random bits and their time bounds are usually *expected*. Randomized dictionaries reached the stage of high development and there is very little left to be improved. On the other hand, deterministic dictionaries with guaranteed time bounds are still evolving. In recent years, a few results in this area have made deterministic dictionaries more competitive. However, many

of these results suffer from *weak non-uniformity* of the algorithms. Namely, the code would require a number of constants to be computed at “compile time” for stated bounds to hold at “run time”. This is a minor flaw if all constants can be computed in time polynomial in  $w$ ; this is the case with fusion trees [6]. But, deterministic dictionaries with lowest known lookup times [9],[10] require constants not known to be computable in polynomial time. This may be a problem even for moderate word sizes and is unacceptable for larger ones. We give an efficient realization of the dictionary which does not suffer from weak non-uniformity.

Fredman, Komlós, and Szemerédi [5] showed that it is possible to construct a linear space dictionary with constant lookup time for arbitrary word sizes. This dictionary implementation is known as the *FKS scheme*. Besides the randomized version they also gave a deterministic construction algorithm with a running time of  $O(n^3w)$ . A bottleneck was the choice of appropriate hash functions. Raman [11] reduced the time for finding a good function to  $O(n^2w)$ . Andersson [1] showed how to modify Raman’s solution to give a time bound  $O(n^{2+\epsilon})$ ; however, he used fusion trees which introduced weak non-uniformity into the solution. Later, Hagerup, Miltersen and Pagh [9] achieved very fast  $O(n \log n)$  construction time for the static case. They combined several techniques, one of which is *error correcting codes*. Yet, the problem of finding a suitable code in polynomial time (posed in [8]) is still unresolved. In the field of large degree trees, which offer balanced lookup and update times, a very efficient structure is described in [2]. It also supports more general predecessor queries.

In this paper we present a family of static and dynamic dictionaries based on a new algorithm for choosing a good hash function. The “quality” of function can be tuned, so various trade-offs may be obtained. A couple of interesting combinations are: search time  $O(1)$  and update time  $O(n^\epsilon \log w)$ ; search time  $O((\log \log n)^2)$  and update time  $o(n^{6/\log \log n} \log w)$ . Mentioned lookup times are not achieved by dictionaries with mild form of weak non-uniformity (polynomial time computable constants). The drawbacks are the presence of  $\log w$  term in update times, and the use of division in dictionary construction.

By introducing (less severe) weak non-uniformity one can remove dependency on  $w$  from update times. For example, our dictionary with search time  $O((\log \log n)^2)$  can be used when  $w = O(n)$ , and exponential search trees [1] can be used for larger  $w$ . This is an alternative to error correcting codes approach. It has higher update times but greater spectrum of applicability (regarding the size of  $w$ ).

Up to now, Raman’s algorithm was the fastest method for finding a suitable hash function with unconstrained range and *constant size* description.<sup>1</sup> We decrease the dependency on  $w$ . Functions usable at the first level of the FKS scheme can be found in time

$$O\left(\min\{n^2 \log^2 n \log w + wn, n^3(\log w + \log n)\}\right).$$

Perfect hash functions can be found in time  $O(n^2 \log^2 n \log w)$ .

<sup>1</sup> Hagerup et al. [9] gave a faster weakly non-uniform algorithm for finding appropriate functions with range  $\Omega(n^2)$ . The method also contains Raman’s construction.

## 2 Family of Functions

Multiplicative hashing families can generally be regarded as families of type:

$$\mathcal{H}_A = \{h_a(x) = \lfloor r * \text{frac}(ax) \rfloor \mid a \in A\} \quad (1)$$

where  $\text{frac}(x)$  denotes  $x - \lfloor x \rfloor$ . Each member of  $\mathcal{H}_A$  maps  $U$  to  $\{0, \dots, r-1\}$ . Functions of this type are used in one of the oldest hashing heuristics known as "the multiplication method". Universal family (shown in [4])

$$\{(ax \bmod 2^w) \operatorname{div} 2^{w-m} \mid a \in U, a \text{ odd}\} \quad (2)$$

is of type (1). Allowing  $a$  to be wider than  $w$  bits, the same holds for

$$\{\lfloor (kx \bmod p) * r/p \rfloor \mid k \in U, p \text{ prime and } p > 2^w\}$$

a variation of the well-known family [5].

For easier notation, by  $Q_v$  we denote  $\{u/2^v \mid 0 < u < 2^v, u \text{ integer}\}$ . The set  $Q_v$  can be regarded as the set of numbers from  $(0, 1)$  given in precision  $2^{-v}$ . Our dictionaries use functions from  $\mathcal{H}_{Q_v}$ , where  $v = O(w)$ . As we will see, discrete families mentioned above inherit good distribution of elements from their real extension  $\mathcal{H}_{\mathbb{R}} = \{h_a : \mathbb{R} \mapsto \mathbf{Z}_r \mid a \in \mathbb{R}\}$ . The following simple result is the base of our construction methods.

**Lemma 1.** *If  $x \neq y$  and*

$$a \in \bigcup_{k \in \mathbf{Z}} \left( \frac{1}{|x-y|} (k+1/r), \frac{1}{|x-y|} (k+1-1/r) \right) \quad (3)$$

*then  $h_a(x) \neq h_a(y)$ , for  $h_a \in \mathcal{H}_{\mathbb{R}}$ .*

*Proof.* For  $h_a(x) \neq h_a(y)$  to be true, it is sufficient that  $|\text{frac}(ax) - \text{frac}(ay)| > 1/r$ . In case  $x > y$  we have

$$\begin{aligned} \text{frac}(a * |x-y|) &= \text{frac}(ax - ay) \\ &= \text{frac}(n_1 + \text{frac}(ax) - (n_2 + \text{frac}(ay))) . \end{aligned}$$

Additionally, if  $\text{frac}(ax) > \text{frac}(ay)$ , the condition becomes  $\text{frac}(a * |x-y|) > 1/r$ . When  $\text{frac}(ax) < \text{frac}(ay)$  we need  $\text{frac}(a * |x-y|) < 1 - \frac{1}{r}$ . The case  $y < x$  makes no difference. It is easy to verify that  $\text{frac}(a * |x-y|) \in (\frac{1}{r}, 1 - \frac{1}{r})$  is equivalent to (3).  $\square$

**Proposition 1.** *Let  $C'$  be the set of all  $a$  from  $[0, 1]$  such that the sum of all pairs  $\{x, y\} \in \binom{S}{2}$  for which  $h_a(x) = h_a(y)$  is less than  $m$ . Then  $C'$  has measure of at least  $1 - \min \left\{ \frac{n(n-1)}{rm}, 1 \right\}$ .*

*Proof.* We denote the intersection of  $[0, 1]$  and the set given in (3) by  $A_{xy}$ . Let  $K_{xy} : [0, 1] \rightarrow \{0, 1\}$  be the characteristic function of the set  $A_{xy}^c$ . The sum of all collisions for a fixed  $a$  is at most  $\sum_{x < y} K_{xy}(a)$  (the condition (3) is not necessary for avoiding the collision between  $x$  and  $y$ ). We use more standard Riemann integral because it is equal to Lebesgue integral in this case.

$$\int_{[0,1]} \sum_{x < y} K_{xy}(t) dt = \sum_{x < y} \int_{[0,1]} K_{xy}(t) dt = \binom{n}{2} \frac{2}{r}$$

The measure of the set  $\{a \in [0, 1] \mid \sum_{x < y} K_{xy}(a) \geq m\}$  can be at most  $\min \left\{ \frac{n(n-1)}{rm}, 1 \right\}$ .  $\square$

From the previous result we also observe that "goodness" of mentioned families is not *essentially* due to some number-theoretic properties; of course, the way a discrete subfamily of  $\mathcal{H}_{\mathbb{R}}$  is chosen does matter. For example, taking  $m = n$  yields functions usable at the first level of the FKS scheme. If the range is set to  $2n$  then more than "half" of functions from  $\mathcal{H}_{\mathbb{R}}$  would be usable. The following variation of Proposition 1 is directly used in Sect. 3. The proof is similar; we only adopt a designation also needed later:  $A_d = A_{xy}$  for  $|x - y| = d$ .

**Lemma 2.** *Let  $D = (d_k)_{k=1}^s$ , where  $d_k$  are elements of  $U \setminus \{0\}$ . Also, let  $m$  be a positive integer,  $B = \{a \in [0, 1] \mid |\{k \mid a \in A_{d_k}^c\}| < m\}$  and*

$$C = \{a \in [0, 1] \mid |\{k \mid (\exists x \in U)(x + d_k \in U \wedge h_a(x) = h_a(x + d_k))\}| < m\} .^2$$

*Then,  $B \subset C$  and the measure of  $B$  is at least  $1 - \min \left\{ \frac{2s}{rm}, 1 \right\}$ .*

We will make an important convention that greatly simplifies notation.  $A_d^c$  consists of disjoint and evenly spaced intervals. We call them support intervals because they form the support of the set's characteristic function. Ends of support intervals are generally not in  $Q_{\lambda w}$ , for an integer constant  $\lambda$  which will be determined by Theorem 1.  $Q_{\lambda w}$  is the set in which we find an appropriate  $a$ . From now on, when we specify some computation involving a support interval, it means that it is performed on the interval obtained by rounding up the support interval's left end, and rounding down the support interval's right end to a number in  $Q_{\lambda w}$ . Some care must be taken – for example, bounds of a support interval cannot be determined by adding  $1/d$  (which is also rounded) to the bounds of the previous interval. The set derived from  $A_d^c$  by rounding its support intervals still contains all unsuitable  $a$ 's from our discrete set. Therefore, all calculated measures will be valid for our goal.

**Lemma 3.** *Given  $d \in U \setminus \{0\}$  and  $b, c \in Q_{\lambda w}$ , the measure of  $[b, c] \cap A_d^c$  can be computed in  $O(1)$  time with use of division.*

<sup>2</sup> Clearly, if  $h_a(x) = h_a(x + d_k)$  for some  $x$  then it holds for all allowed  $x$ .

*Proof.* Let support intervals of  $A_d^c$  be  $[b_k, c_k]$ , where  $0 \leq k \leq d$ . Let  $i = \max\{k \mid b_k < b\}$  and  $s = \min\{k \mid c_k > c\}$ ; they can be determined in constant time – we use division by  $1/d$ , which amounts to multiplication. If  $i = s$  the result is  $c - b$ . Otherwise, we compute the lengths of  $[b, c] \cap [b_i, c_i]$  and  $[b, c] \cap [b_s, c_s]$ . To their sum we then add  $\frac{2(s-i-1)}{rd}$  and get the measure. The division can be accomplished in constant time since both operands are  $O(w)$  bit integers.  $\square$

**Lemma 4.** *Let  $B$  be the set defined in Lemma 2. If  $c - b \leq 2^{-w - \lceil \lg r \rceil}$ , then  $B \cap [b, c]$  consists of at most  $m$  different intervals.*

*Proof.* Only one support interval from each set  $A_d^c$  can have nonempty intersection with  $[b, c]$  and it cannot be completely inside  $[b, c]$  because it has greater length. Let  $(c_k)_{k=1}^p$ ,  $p \leq \binom{n}{2}$ , be the increasing sequence of right ends of specified intervals that have nonempty intersection with  $[b, c]$ . Also let  $q = \max\{0, p - m + 1\}$  and  $c_0 = b$ . Then  $[b, c_q]$  is not in  $B$ . Suppose that  $[a_1, a_2]$  is not in  $B$  and that  $(a_2, a_3)$  is. The number of collisions decreases at  $a_2$  and this can happen only at one of the points  $c_{q+i}$ ,  $0 \leq i < m$ . Thus, there are at most  $m$  left boundary points for  $B \cap [b, c]$ , so it consists of at most  $m$  different intervals.  $\square$

### 3 Finding a Good Function

The main algorithm, described in Theorem 1, accepts as input a finite sequence of values from  $\{|x - y| \mid x, y \in S\}$ . For simplicity we assume that the elements of the sequence are distinct; therefore, we may use set notation. If a value occurs several times within the sequence, it is processed every time; no part of further discussion is affected by this.

We denote input set (sequence) of differences by  $D$ . The algorithm needs to process elements of  $D$  in increasing order several times. Storing  $D$  in memory would generally be unacceptable. However, the types of subsets of  $\{|x - y| \mid x, y \in S\}$  used by our dictionaries can be enumerated in increasing order using only  $O(n)$  space. In other words, after calling a function that initializes some common variables, another function could serve elements of  $D$  one by one, as claimed by the next lemma.

**Lemma 5.** *Let  $S = \{x_1, x_2, \dots, x_n\}$ , with  $x_i < x_{i+1}$ . Further, let  $(n_k)_{k=1}^n$  be a given sequence of positive integers,  $n_k \leq n/2$ , and*

$$D = \bigcup_{k=1}^n \bigcup_{j=1}^{n_k} \{|x_{k+n_j} - x_k|\}.^3 \quad (4)$$

*Using linear space we can go through the elements of  $D$  in increasing order, with each retrieval taking time  $O(\log n)$ .*

<sup>3</sup> By  $x +_n y$  we denote  $[(x + y - 1) \bmod n] + 1$ . For convenience we set the range of the function to be  $\{1, \dots, n\}$ .

*Proof.* At the beginning, the elements of  $S$  are sorted. Candidates for the minimum of  $D$  are differences of type  $|x_{k+n_1} - x_k|$  and  $|x_k - x_{k+n_k}|$ . Initially, for each  $x_k$  the smaller of the two is inserted into a priority queue. Each difference is accompanied by the index of the element which the difference is attributed to. Further, for each  $x_k$  two indices are maintained, call them  $l_k$  and  $r_k$ . They mean that  $|x_k - x_{l_k}|$  and  $|x_{r_k} - x_k|$  are the last “left” and “right” difference attributed to  $x_k$  that were inserted into the queue. After we fetch currently the smallest difference, suppose that is  $|x_{r_j} - x_j|$ , we take smaller of the next “left” and “right” differences related to  $x_j$  and insert it into the queue. At all times there are  $O(n)$  elements in the heap. Indices  $l_j$  and  $r_j$  are not wrapped when they hit their bounds, 1 and  $n$ .

Simple binary heap can be used for the priority queue because there would be no considerable gain from using something more sophisticated. Similarly, a simple  $O(n \log n)$  sorting algorithm is adequate.  $\square$

Note that the state of the process can be saved at some desired time during iteration so we don’t have to start from the beginning every time. In the following theorem we implicitly assume that input set  $D$  is enumerable as we described.

**Theorem 1.** *Let  $D = \{d_1, d_2, \dots, d_s\}$ , where  $d_k$  are elements of  $U \setminus \{0\}$ , and let  $v = w + \lceil \lg m \rceil + \lceil \lg r \rceil + 3$ . If  $rm \geq 4s$ ,<sup>4</sup> we can find  $a \in Q_v \cap B$  ( $B$  is defined in Lemma 2) with a deterministic algorithm that uses space  $O(n + m)$  and runs in time*

$$\min \left\{ \frac{O(s \log n (\log w + \log r) + s \log m + wm)}{O(s (\log w + \log r) (m + \log n))} \right\}. \quad (5)$$

*Proof.* We first use bit by bit construction to sufficiently narrow the interval in which we seek for  $a$ , and then we explicitly determine the intersection of  $B$  and that small interval. On some events several bits may be set at once.

On start no bits are selected and the active interval is  $(0, 1)$ . Suppose we came down to the interval  $(a_1, a_2)$  and that we have selected  $p$  most significant bits. We partition the set of differences into three classes:  $D = D_{\text{big}} \cup D_{\text{mid}} \cup D_{\text{sm1}}$  where  $D_{\text{big}} = \{d \in D \mid \frac{2}{rd} \geq a_2 - a_1\}$ ,  $D_{\text{sm1}} = \{d \in D \mid \frac{2}{rd} < \frac{a_2 - a_1}{2(w + \log r)r}\}$  and  $D_{\text{mid}} = D \setminus (D_{\text{big}} \cup D_{\text{sm1}})$ . Smaller  $d$  corresponds to bigger support interval.

We maintain information about  $(a_1, a_2) \cap A_d^c$  for all  $d \in D_{\text{big}}$ . If the intersection is nonempty, it is completely determined either by the right end of a support interval, or the left end, or  $(a_1, a_2) \in A_d^c$ . Support intervals that cover  $(a_1, a_2)$  are irrelevant to the choices made by the algorithm. For accounting partial intersections, two ordered sequences are stored – one for right ends and one for left ends. Observe that we need information only about the last  $m$  right ends and the first  $m$  left ends. Let  $b_1$  be the  $m$ -th largest right end and  $b_2$  be the  $m$ -th smallest left end (if they don’t exist take  $b_1 = a_1$  and/or  $b_2 = a_2$ ). The elements of  $(a_1, b_1]$  and  $[b_2, a_2)$  are certainly not in  $B$ . We can employ, for example, a

<sup>4</sup> A theorem with requirement  $\frac{2s}{rm} \leq \nu < 1$ , for constant  $\nu$ , could similarly be proved. Only constants hidden in the time bound and the word length of  $a$  would be affected.

B-tree structure to maintain this information. If the number of elements in the tree that stores right ends grows to  $m + 1$ , the smallest element is deleted as it becomes irrelevant. The same goes for the tree that holds left ends, except that the largest element gets deleted. The structure is updated as  $p$  increases.

Let  $b = (a_1 + a_2)/2$ . We estimate  $m(B^c \cap (a_1, b))$  and  $m(B^c \cap (b, a_2))$ ,<sup>5</sup> and the interval with lower value (greater estimated intersection with  $B$ ) wins;  $p$ -th bit is set accordingly (bit positions are zero based). From the requirements of the theorem and our construction, it will follow that  $B \cap (a_1, a_2) \neq \emptyset$ ; hence, it must be  $b_1 < b_2$ . If  $b_2 \leq b$  then  $(a_1, b)$  is chosen, if  $b_1 \geq b$  then  $(b, a_2)$  is chosen. Now, suppose that  $b_1 < b < b_2$ . Like in the proof of Lemma 2 (Proposition 1):

$$\begin{aligned} m(B^c \cap (a_1, b)) &= m((a_1, b_1]) + m(B^c \cap (b_1, b)) \\ &\leq b_1 - a_1 + \frac{1}{m} \sum_{k=1}^s m(A_{d_k}^c \cap (b_1, b)) \\ &= b_1 - a_1 + \frac{1}{m} \left( \sum_{d \in D_{\text{big}}} m(A_d^c \cap (b_1, b)) + \right. \\ &\quad \left. \sum_{d \in D_{\text{mid}}} m(A_d^c \cap (b_1, b)) + \sum_{d \in D_{\text{small}}} m(A_d^c \cap (b_1, b)) \right). \end{aligned}$$

The sum over the elements of  $D_{\text{big}}$  can be computed in time  $O(m)$ . The terms in the middle sum can be computed in constant time per set (Lemma 3). The last sum can be estimated by  $\frac{1}{m}|D_{\text{small}}|(b - b_1)\frac{2}{r}$ . The error produced by this estimate can be bounded by using the upper bound on the length of one support interval per each member of  $D_{\text{small}}$ :

$$E = \frac{1}{m}|D_{\text{small}}|\frac{a_2 - a_1}{2(w + \log r)r} \leq \frac{s(a_2 - a_1)}{2(w + \log r)rm}.$$

Summing all obtained values, we get an estimate for  $m(B^c \cap (a_1, b))$  – denote it by  $\mu_1^{(p+1)}$ . The other interval  $(b, a_2)$  is processed analogously and resulting estimate is denoted by  $\mu_2^{(p+1)}$ . Note that the error bound  $E$  is the sum of errors made on both intervals because the intervals are consecutive.

The same kind of measure estimate performed for  $(a_1, a_2)$  (in the previous step) is denoted by  $\mu^{(p)}$  – the value corresponding to the winning interval in  $p$ -th step. Similarly for  $E^{(p)}$ . Since we half the active interval in each step, it is  $a_2 - a_1 = 2^{-p}$ . Observe that the difference  $(\mu^{(p)} + E^{(p)}) - m(B^c \cap (a_1^{(p)}, a_2^{(p)}))$  (relative to  $2^{-p}$ ) decreases as  $p$  increases, because elements fall out of  $D_{\text{small}}$  and fall in  $D_{\text{big}}$  (not directly, though). Namely:

$$\mu_1^{(p+1)} + \mu_2^{(p+1)} + E^{(p+1)} \leq \mu^{(p)} + E^{(p)}.$$

Setting  $\mu^{(p+1)} = \min\{\mu_1^{(p+1)}, \mu_2^{(p+1)}\}$  gives the recurrence

$$\mu^{(p+1)} \leq \frac{1}{2} \left( \mu^{(p)} + E^{(p)} \right) \leq \frac{1}{2} \left( \mu^{(p)} + \frac{1}{2^{p+1}(w + \log r)} \frac{s}{rm} \right).$$

<sup>5</sup> Do not confuse  $m(X)$  as the measure set function and  $m$  as the number of allowed collisions.

An upper bound on  $\mu^{(p+1)}$  is then

$$\begin{aligned}\mu^{(p+1)} &\leq \frac{1}{2^{p+1}}\mu^{(0)} + \sum_{i=0}^p \frac{1}{2^{p+1-i}} \frac{1}{2^{i+1}(w + \lg r)} \frac{s}{rm} \\ &= \frac{1}{2^{p+1}}\mu^{(0)} + \frac{p}{2^{p+3}(w + \lg r)} \frac{2s}{rm} .\end{aligned}\quad (6)$$

Initial bound for  $m(B^c \cap (0, 1))$  is given by Lemma 2:  $\mu^{(0)} = \frac{2s}{rm} \leq 1/2$ . Bit by bit construction i.e. narrowing of the active interval is done until  $p = w + \lceil \lg r \rceil \equiv q$ . Substituting in (6) gives

$$\begin{aligned}m(B^c \cap (a_1^{(q)}, a_2^{(q)})) &\leq \mu^{(q)} + E^{(q)} \\ &\leq 2^{-q-1} + 2^{-q-3} + 2^{-q-3} = \frac{3}{4}m((a_1^{(q)}, a_2^{(q)})) .\end{aligned}\quad (7)$$

When  $p = q$ , then  $D = D_{\text{big}}^{(p)}$ . From Lemma 4 and (7) it follows that within  $B \cap (a_1^{(q)}, a_2^{(q)})$  there must be an interval of length at least  $\frac{1}{4m}2^{-q}$ . Hence, using the B-trees, in time  $O(m)$  we can find such interval and within it an element from  $Q_{w+\lceil \lg r \rceil + \lceil \lg m \rceil + 3}$ .

Non-constant space requirements come from the B-trees and the heap used for retrieving members of  $D$ . They need space  $O(m)$  and  $O(n)$ , respectively. It remains to show the total time bound. There are  $O(w)$  bit choosing steps and elements from  $D_{\text{sml}}$  are never considered. Elements from  $D_{\text{mid}}^{(p)}$  are processed in time  $O(|D_{\text{mid}}^{(p)}| \log n)$ , where  $\log n$  factor comes from retrieving a difference from the heap. We show that an element of  $D$  can stay in  $D_{\text{mid}}$  at most  $O(\log w + \log r)$  steps. Element  $d$  is in  $D_{\text{mid}}$  iff

$$\begin{aligned}\frac{1}{2^{p+1}(w + \lg r)r} &< \frac{2}{rd} < \frac{1}{2^p} \iff O(w) * 2^{p+2} > d > \frac{2^{p+1}}{r} \\ &\iff \lg d - O(\lg w) < p < \lg d + O(\lg r) .\end{aligned}$$

Each element eventually enters  $D_{\text{big}}$  and update time per element is  $O(\log m)$ . Thus, the total load coming from  $D_{\text{mid}}$  is  $O(s \log n (\log w + \log r) + s \log m)$ .

In any step for which  $D_{\text{mid}}^{(p)} \neq \emptyset$ , we use the tree structures to acquire the contribution of elements from  $D_{\text{big}}^{(p)}$  to  $\mu_{1/2}^{(p)}$ . This takes  $O(m)$  time since we process elements sequentially; the procedure occurs at most

$$\min\{O(w), O(s(\log w + \log r))\}$$

times. Finally, consider the case  $D_{\text{mid}}^{(p)} = \emptyset$ . We prefetch next  $d_k$  and see in which step will it be processed first. That is, we find the smallest  $p_1 > p$  such that  $D_{\text{mid}}^{(p_1)} \neq \emptyset$ .  $p_1$  can be computed in time  $O(\log w)$  and this is done at most once per element of  $D$ . Next, find an appropriate interval of length  $2^{-p_1}$  with ends in  $Q_{p_1}$ , which has minimal  $\mu^{(p_1)}$ . This takes  $O(m)$  time – an observation similar to the one in the proof of Lemma 4 is needed. Multiple bits are set at once if  $p_1 > p + 1$ .  $D_{\text{mid}}^{(p)}$  can be empty at most  $\min\{O(w), s\}$  times. Adding all the bounds proves the theorem.  $\square$



Remark that the algorithm itself is not too complicated, in spite of lengthy justification of its actions. Auxiliary structures used are the classical priority queue and the slightly modified B-tree. As a result, aggregate implementation complexity is not high. Use of some more efficient structures would make no significant gain. The disadvantages of the algorithm are the use of division and the bit length of result value  $a$ . For typical relations of parameters  $w, m, r$  the evaluation of the function  $h_a$  would require 2-3 multiplications.

## 4 Uniform Dictionaries

**Theorem 2.** *There is a linear space static dictionary with query time  $O(1)$  and with deterministic construction time  $O(n^{1+\epsilon} \log w)$ , for any fixed  $\epsilon > 0$ .*

*Proof.* We employ multi-level hashing scheme. Choose some  $\epsilon_1$  such that  $0 < \epsilon_1 < \epsilon/2$ . Initially, set  $n_k = \lceil n^{\epsilon_1} \rceil$ ,  $1 \leq k \leq n$  (recall Lemma 5) and set the range to  $r = n$ . Also, we allow  $m = 4\lceil n^{\epsilon_1} \rceil$  collisions, thereby satisfying the condition of Theorem 1 that  $\frac{2s}{rm} \leq 1/2$ . The algorithm described in Theorem 1 returns value  $a$ . We assume the time bound that contains only  $\log w$  factor (does not have factor  $w$ ). Since  $n^{2\epsilon_1} \log n = o(n^\epsilon)$  a time bound for the first level is  $O(n^{1+\epsilon} \log w)$ .

Define  $P_x = \{y \in S \mid |x - y| \in D \wedge h_a(x) = h_a(y)\}$ ;  $|\bigcup_{k=1}^n P_{x_k}| < m$ . Suppose that  $h_a(x) = z$ , for some  $x \in S$ . At least  $n^{\epsilon_1} - |P_x|$  keys cannot hash to slot  $z$ . Take  $y \in S$ , such that  $y \notin P_x$ ,  $x \notin P_y$  and  $h_a(y) = z$ . Then, *another*<sup>6</sup>  $n^{\epsilon_1} - |P_y|$  keys cannot hash to slot  $z$ . Utilizing an induction argument we get an upper bound on the longest chain:  $n^{1-\epsilon_1} + m$ . Without l.o.g. we may assume  $\epsilon_1 < 1/2$ . The length of the longest chain becomes  $O(n^{1-\epsilon_1})$ . If the size of a bucket is less than  $n^{\epsilon_1}$ , the parameter settings  $(r, m)$  of the FKS scheme are chosen and the bucket is resolved in two steps. Otherwise, the above procedure is recursively applied to the bucket. It takes at most  $\left\lceil \frac{\log \epsilon_1}{\log(1-\epsilon_1)} \right\rceil$  levels till the employment of the FKS scheme.

If  $C(N)$  is the total construction time for a bucket of size  $N$  *excluding* the times for FKS constructions, it is bounded by recurrence inequality

$$\begin{aligned} C(N) &\leq O(N^{1+\epsilon} \log w) + N^{\epsilon_1} C(N^{1-\epsilon_1}) \\ \Rightarrow C(N) &= O(N^{1+\epsilon} \log w) . \end{aligned}$$

The total time for the FKS constructions is at most

$$n^{1-\epsilon_1} O(n^{3\epsilon_1} (\log w + \log n)) = O(n^{1+\epsilon} \log w) .$$

□

The static dictionary described in Theorem 2 can be dynamized the same way as described in [8].

<sup>6</sup> This follows from the structure of  $D$ , given by (4).

**Theorem 3.** *Let  $t : \mathbb{N} \mapsto \mathbb{N}$  be a nondecreasing function computable in time and space  $O(n)$ , and  $4 < t(n) \leq O(\sqrt{\log n} / \log \log n)$ . There exists a linear space static deterministic dictionary with query time  $O(t(n) \log \log n)$  and construction time*

$$O(n^{1+2/t(n)} t^2(n) \log n \log \log n (\log w + \log n)) .$$

*Proof.* Construction proceeds in the manner of Theorem 2, except that we don't utilize the FKS scheme – the construction is continued until bucket size drops below  $\log n$ . Comparison based methods finish the search. The setting of the parameters for a bucket of size  $N$  is:

$$s = N^{1+1/t(n)}, \quad r = \frac{N}{t(n) \log \log n}, \quad m = 4N^{1/t(n)} t(n) \log \log n .^7$$

For the longest chain to be  $O(N^{1-1/t(n)})$ , it must hold  $m = O(N^{1-1/t(n)})$ , which is satisfied due to  $N \geq \log n$  and the condition  $4 < t(n) \leq O(\sqrt{\log n} / \log \log n)$ .

The number of levels is determined by  $L(N) = 1 + L(N^{1-1/t(n)})$ . The condition  $t(n) > 4$  ensures that  $|\log(1 - 1/t(n))| = O(1/t(n))$ . Hence

$$L(N) = O(t(n) \log \log N) ,$$

which also bounds the search time.

The full construction time is given by

$$\begin{aligned} C(N) &\leq O(N^{1+1/t(n)} (\log w + \log n) (m + \log n)) + N^{1/t(n)} C(N^{1-1/t(n)}) \\ &\Rightarrow C(N) \leq O(N^{1+1/t(n)} (\log w + \log n) N^{1/t(n)} t(n) \log n) + \\ &\quad N^{1/t(n)} C(N^{1-1/t(n)}) \\ &\Rightarrow C(N) = L(N) * O(N^{1+2/t(n)} t(n) \log n (\log w + \log n)) . \end{aligned}$$

The setting for  $r$  guarantees linear space consumption. □

The described static dictionary is dynamized naturally because data is stored in a tree structure.

**Theorem 4.** *Let  $t : \mathbb{N} \mapsto \mathbb{N}$  be a nondecreasing function computable in time and space  $O(n)$ , and  $4 < t(n) \leq O(\sqrt{\log n} / \log \log n)$ . There exists a linear space dynamic deterministic dictionary with query time  $O(t(n) \log \log n)$  and update time*

$$O(n^{5/t(n)} t^2(n) \log n \log \log n (\log w + \log n)) .$$

*The bound for lookup time is worst case, while the bound for updates time is amortized.*

---

<sup>7</sup> We omitted ceilings for clarity.

*Proof.* Standard rebuilding scheme of varying locality is used. Rebuilding initiated at some node causes the reconstruction of the whole subtree rooted at that node. The local reconstruction of a subtree is performed after the number of updates reaches the half of the number of elements present in that subtree on the last rebuilding. Assume that only the insertions are performed; this case is worse than the case of mixed insertions and deletions because the size changes faster. Also assume that new elements are “pumped” along the same path of the global tree. It will be clear that this is the worst case.

Suppose that the number of elements in a subtree was  $N$  after the last rebuilding which had occurred at some higher level node – an ancestor of the subtree’s root. We want to bound the total time spent on rebuildings of the subtree *between* rebuildings initiated at higher level nodes. The settings of the construction parameters  $(s, r, m)$  depend on the value of  $n$ . In the dynamic case,  $n$  is the total number of elements on the last global rebuilding. Let  $K_n = t^2(n) \log n \log \log n (\log w + \log n)$  and  $t = t(n)$ . The sum of times for the reconstructions of the whole subtree is

$$O \left( \left[ \frac{3}{2} N \right]^{1+2/t} K_n + \left[ \left( \frac{3}{2} \right)^2 N \right]^{1+2/t} K_n + \dots + \left[ \left( \frac{3}{2} \right)^p N \right]^{1+2/t} K_n \right) \quad (8)$$

where  $p$  is such that  $\left(\frac{3}{2}\right)^{p+1} N = \frac{1}{2} N^{t/(t-1)} + N$ ; the right side represents the number of updates, and  $N^{t/(t-1)}$  is the number of elements contained in the tree rooted at the direct ancestor node of the subtree we observe. We used equal sign in the definition of  $p$  because, for easier proof, we assume that  $p$  is an integer. The assumption does not affect the asymptotic bounds. Putting  $q = \left(\frac{3}{2}\right)^{1+2/t}$ , (8) becomes

$$\begin{aligned} & O(N^{1+2/t} K_n (q + q^2 + \dots + q^p)) \\ &= O \left( N^{1+2/t} K_n q \frac{q^p - 1}{q - 1} \right) = O \left( N^{1+2/t} K_n q^p \right) . \end{aligned} \quad (9)$$

From the definitions of  $p$  and  $q$  we get

$$q^{p+1} = \left( \frac{1}{2} N^{1/(t-1)} + 1 \right)^{1+2/t} = O(N^{\frac{t+2}{t(t-1)}}) .$$

Using  $t(n) > 4$  and substituting into (9) gives a bound  $O(N^{1+4/t(n)} K_n)$ . In order to find the complete sum of times, we must include reconstructions initiated at lower level nodes descending from the subtree’s root. Let  $R(N)$  denote the desired time. As we said,  $N$  is the size of the subtree at the beginning of the phase which ends at a higher level reconstruction. The initial size of a bucket in the root node is  $N^{1-1/t}$ , but this value grows after each reconstruction of the

root. Thus, the following recurrence holds:

$$R(N) = O(N^{1+4/t}K_n) + R(N^{1-1/t}) + R\left(\left\lceil\frac{3}{2}N\right\rceil^{1-1/t}\right) + \dots \\ + R\left(\left[\left(\frac{3}{2}\right)^p N\right]^{1-1/t}\right).$$

The last term in the sum is less than  $R(N/3)$ . An induction argument proves that  $R(N) = O(N^{1+6/t}K_n)$ . The total time for  $n/2$  insertions includes a global rebuilding:

$$R(n^{1-1/t(n)}) + O\left(\left(\frac{3}{2}n\right)^{1+2/t(n)}K_n\right) = O(n^{1+5/t(n)}K_n).$$

□

## References

1. Arne Andersson. Faster deterministic sorting and searching in linear space. In Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS '96), pages 135–141.
2. Paul Beame and Faith Fich. Optimal bounds for the predecessor problem. In Proceedings of the 31th Annual ACM Symposium on Theory of Computing (STOC '99), pages 295–304.
3. Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.
4. Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.
5. Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with  $O(1)$  worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538–544, 1984.
6. Michael L. Fredman and Dan E. Willard. Surpassing the information-theoretic bound with fusion trees. *J. Comput. System Sci.*, 47:424–436, 1993.
7. Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley Publishing Co., Reading, Mass., 1973. Volume 3. Sorting and searching.
8. Peter Bro Miltersen. Error correcting codes, perfect hashing circuits, and deterministic dynamic dictionaries. In Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998), pages 556–563, ACM Press, 1998.
9. Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic Dictionaries. *Journal of Algorithms* 41(1):69–85, 2001.
10. Rasmus Pagh. A trade-off for worst-case efficient dictionaries. *Nordic Journal of Computing* 7(3):151–163, 2000.
11. Rajeev Raman. Priority queues: small, monotone and transdichotomous. In Proceedings of the 4th European Symposium on Algorithms (ESA '96), pages 121–137, 1996.
12. Robert E. Tarjan and Andrew Chi Chih Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.

# Fast Sparse Matrix Multiplication

Raphael Yuster<sup>1</sup> and Uri Zwick<sup>2</sup>

<sup>1</sup> Department of Mathematics, University of Haifa at Oranim, Tivon 36006, Israel.

<sup>2</sup> School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel

**Abstract.** Let  $A$  and  $B$  two  $n \times n$  matrices over a ring  $R$  (e.g., the reals or the integers) each containing at most  $m$  non-zero elements. We present a new algorithm that multiplies  $A$  and  $B$  using  $O(m^{0.7}n^{1.2} + n^{2+o(1)})$  algebraic operations (i.e., multiplications, additions and subtractions) over  $R$ . For  $m \leq n^{1.14}$ , the new algorithm performs an almost optimal number of only  $n^{2+o(1)}$  operations. For  $m \leq n^{1.68}$ , the new algorithm is also faster than the best known matrix multiplication algorithm for dense matrices which uses  $O(n^{2.38})$  algebraic operations. The new algorithm is obtained using a surprisingly straightforward combination of a simple combinatorial idea and existing fast *rectangular* matrix multiplication algorithms. We also obtain improved algorithms for the multiplication of more than two sparse matrices. As the known fast rectangular matrix multiplication algorithms are far from being practical, our result, at least for now, is only of theoretical value.

## 1 Introduction

The multiplication of two  $n \times n$  matrices is one of the most basic algebraic problems and considerable effort was devoted to obtaining efficient algorithms for the task. The naive matrix multiplication algorithm performs  $O(n^3)$  operations. Strassen [23] was the first to show that the naive algorithm is not optimal, giving an  $O(n^{2.81})$  algorithm for the problem. Many improvements then followed. The currently fastest matrix multiplication algorithm, with a complexity of  $O(n^{2.38})$ , was obtained by Coppersmith and Winograd [8]. More information on the fascinating subject of matrix multiplication algorithms and its history can be found in Pan [16] and Bürgisser *et al.* [3]. An interesting new group theoretic approach to the matrix multiplication problem was recently suggested by Cohn and Umans [6]. For the best available lower bounds see Shpilka [22] and Raz [18].

Matrix multiplication has numerous applications in combinatorial optimization in general, and in graph algorithms in particular. Fast matrix multiplication algorithms can be used, for example, to obtain fast algorithms for finding simple cycles in graphs [1,2,24], for finding small cliques and other small subgraphs [15], for finding shortest paths [20,21,25], for obtaining improved dynamic reachability algorithms [9,19], and for matching problems [14,17,5]. Other applications can be found in [4,13], and this list is not exhaustive.

In many cases, the matrices to be multiplied are *sparse*, i.e., the number of non-zero elements in them is negligible compared to the number of zeros in them. For example if  $G = (V, E)$  is a directed graph on  $n$  vertices containing  $m$  edges, then its adjacency matrix  $A_G$  is an  $n \times n$  matrix with  $m$  non-zero elements (1's in this case). In many interesting cases  $m = o(n^2)$ . Unfortunately, the fast matrix multiplication algorithms mentioned above cannot utilize the sparsity of the matrices multiplied. The complexity of the algorithm of [8], for example, remains  $O(n^{2.38})$  even if the multiplied matrices are extremely sparse. The naive matrix multiplication algorithm, on the other hand, can be used to multiply two  $n \times n$  matrices, each with at most  $m$  non-zero elements, using  $O(mn)$  operations (see next section). Thus, for  $m = O(n^{1.37})$ , the sophisticated matrix multiplication algorithms do not provide any improvement over the naive matrix multiplication algorithm.

In this paper we show that the sophisticated matrix multiplication algorithms *can* nevertheless be used to speed-up the computation of the product of even extremely sparse matrices. More specifically, we present a new algorithm that multiplies two  $n \times n$  matrices, each with at most  $m$  non-zero elements, using  $O(m^{0.7}n^{1.2} + n^{2+o(1)})$  algebraic operations. (The exponents 0.7 and 1.2 are derived, of course, from the current 2.38 bound on the exponent of matrix multiplication, and from bounds on other exponents related to matrix multiplications, as will be explained in the sequel.) There are three important things to notice:

- (i) If  $m \geq n^{1+\epsilon}$ , for any  $\epsilon > 0$ , then the number of operations performed by the new algorithm is  $o(mn)$ , i.e., less than the number of operations performed, in the worst-case, by the naive algorithm.
- (ii) If  $m \leq n^{1.14}$ , then the new algorithm performs only  $n^{2+o(1)}$  operations. This is very close to optimal as all  $n^2$  entries in the product may be non-zero, even if the multiplied matrices are very sparse.
- (iii) If  $m \leq n^{1.68}$ , then the new algorithm performs only  $o(n^{2.38})$ , i.e., fewer operations than the fastest known matrix multiplication algorithm.

In other words, the new algorithm improves on the naive algorithm even for extremely sparse matrices (i.e.,  $m = n^{1+\epsilon}$ ), and it improves on the fastest matrix multiplication algorithm even for relatively dense matrices (i.e.,  $m = n^{1.68}$ ).

The new algorithm is obtained using a surprisingly straightforward combination of a simple combinatorial idea, implicit in Eisenbrand and Grandoni [10] and Yuster and Zwick [24], with the fast matrix multiplication algorithm of Coppersmith and Winograd [8], and the fast *rectangular* matrix multiplication algorithm of Coppersmith [7]. It is interesting to note that a fast rectangular matrix multiplication algorithm for dense matrices is used to obtain a fast matrix multiplication algorithm for sparse square matrices.

As mentioned above, matrix multiplication algorithms are used to obtain fast algorithms for many different graph problems. We note (with some regret . . .) that our improved sparse matrix multiplication algorithm does not yield, automatically, improved algorithms for these problems on sparse graphs. These algorithms may need to multiply dense matrices even if the input graph is sparse. Consider

for example the computation of the transitive closure of a graph by repeatedly squaring its adjacency matrix. The matrix obtain after the first squaring may already be extremely dense. Still, we expect to find many situations in which the new algorithm presented here could be useful.

In view of the above remark, we also consider the problem of computing the product  $A_1 A_2 \cdots A_k$  of three or more sparse matrices. As the product of even very sparse matrices can be completely dense, the new algorithm for multiplying two matrices cannot be applied directly in this case. We show, however, that some improved bounds may also be obtained in this case. Our results here are less impressive, however. For  $k = 3$ , we improve, for certain densities, on the performance of all existing algorithms. For  $k \geq 4$  we get no worst-case improvements at the moment, but such improvements will be obtained if bounds on certain matrix multiplication exponents are sufficiently improved.

The rest of the paper is organized as follows. In the next section we review the existing matrix multiplication algorithms. In Section 3 we present the main result of this paper, i.e., the improved sparse matrix multiplication algorithm. In Section 4 we use similar ideas to obtain an improved algorithm for the multiplication of three or more sparse matrices. We end, in Section 5, with some concluding remarks and open problems.

## 2 Existing Matrix Multiplication Algorithms

In this short section we examine the worst-case behavior of the naive matrix multiplication algorithm and state the performance of existing fast matrix multiplication algorithms.

### 2.1 The Naive Matrix Multiplication Algorithm

Let  $A$  and  $B$  be two  $n \times n$  matrices. The product  $C = AB$  is defined as follows  $c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$ , for  $1 \leq i, j \leq n$ . The naive matrix multiplication algorithm uses this definition to compute the entries of  $C$  using  $n^3$  multiplications and  $n^3 - n^2$  additions. The number of operations can be reduced by avoiding the computation of products  $a_{ik} b_{kj}$  for which  $a_{ik} = 0$  or  $b_{kj} = 0$ . In general, if we let  $\bar{a}_k$  be the number of non-zero elements in the  $k$ -th column of  $A$ , and  $\bar{b}_k$  be the number of non-zero elements in the  $k$ -th row of  $B$ , then the number of multiplications that need to be performed is only  $\sum_{k=1}^n \bar{a}_k \bar{b}_k$ . The number of additions required is always bounded by the required number of multiplications. This simple sparse matrix multiplication algorithm may be considered folklore. It can also be found in Gustavson [11].

If  $A$  contains at most  $m$  non-zero entries, then  $\sum_{k=1}^n \bar{a}_k \bar{b}_k \leq (\sum_{k=1}^n \bar{a}_k) n \leq mn$ . The same bound is obtained when  $B$  contains at most  $m$  non-zero entries. Can we get an improved bound on the worst-case number of products required when both  $A$  and  $B$  are sparse? Unfortunately, the answer is no. Assume that

$m \geq n$  and consider the case  $\bar{a}_i = \bar{b}_i = n$ , if  $i \leq m/n$ , and  $\bar{a}_i = \bar{b}_i = 0$ , otherwise. (In other words, all non-zero elements of  $A$  and  $B$  are concentrated in the first  $m/n$  columns of  $A$  and the first  $m/n$  columns of  $B$ .) In this case  $\sum_{k=1}^n \bar{a}_k \bar{b}_k = (m/n) \cdot n^2 = mn$ . Thus, the naive algorithm may have to perform  $mn$  multiplications even if both matrices are sparse. It is instructive to note that the computation of  $AB$  in this worst-case example can be reduced to the computation of a much smaller rectangular product. This illustrates the main idea behind the new algorithm: When the naive algorithm performs many operations, rectangular matrix multiplication can be used to speed up the computation.

To do justice with the naive matrix multiplication algorithm we should note that in many cases that appear in practice the matrices to be multiplied have a special structure, and the number of operations required may be much smaller than  $mn$ . For example, if the non-zero elements of  $A$  are evenly distributed among the columns of  $A$ , and the non-zero elements of  $B$  are evenly distributed among the rows of  $B$ , we have  $\bar{a}_k = \bar{b}_k = m/n$ , for  $1 \leq k \leq n$ , and  $\sum_{k=1}^n \bar{a}_k \bar{b}_k = n \cdot (m/n)^2 = m^2/n$ . We are interested here, however, in worst-case bounds that hold for any placement of non-zero elements in the input matrices.

## 2.2 Fast Matrix Multiplication Algorithms for Dense Matrices

Let  $M(a, b, c)$  be the minimal number of algebraic operations needed to multiply an  $a \times b$  matrix by a  $b \times c$  matrix over an arbitrary ring  $R$ . Let  $\omega(r, s, t)$  be the minimal exponent  $\omega$  for which  $M(n^r, n^s, n^t) = O(n^{\omega+o(1)})$ . We are interested here mainly in  $\omega = \omega(1, 1, 1)$ , the exponent of square matrix multiplication, and  $\omega(1, r, 1)$ , the exponent of rectangular matrix multiplication of a particular form. The best bounds available on  $\omega(1, r, 1)$ , for  $0 \leq r \leq 1$  are summarized in the following theorems:

**Theorem 1 (Coppersmith and Winograd [8]).**  $\omega < 2.376$ .

We next define the constants  $\alpha$  and  $\beta$  of rectangular matrix multiplication.

**Definition 1.**  $\alpha = \max\{0 \leq r \leq 1 \mid \omega(1, r, 1) = 2\}$ ,  $\beta = \frac{\omega - 2}{1 - \alpha}$ .

**Theorem 2 (Coppersmith [7]).**  $\alpha > 0.294$ .

It is not difficult to see that these Theorems 1 and 2 imply the following theorem. A proof can be found, for example, in Huang and Pan [12].

**Theorem 3.**  $\omega(1, r, 1) \leq \begin{cases} 2 & \text{if } 0 \leq r \leq \alpha, \\ 2 + \beta(r - \alpha) & \text{otherwise.} \end{cases}$

**Corollary 1.**  $M(n, \ell, n) \leq n^{2-\alpha\beta+o(1)}\ell^\beta + n^{2+o(1)}$ .

All the bounds in the rest of the paper will be expressed terms of  $\alpha$  and  $\beta$ . Note that with  $\omega = 2.376$  and  $\alpha = 0.294$  we get  $\beta \simeq 0.533$ . If  $\omega = 2$ , as conjectured by many, then  $\alpha = 1$ . (In this case  $\beta$  is not defined, but also not needed.)



### 3 The New Sparse Matrix Multiplication Algorithm

Let  $A_{*k}$  be the  $k$ -th column of  $A$ , and let  $B_{k*}$  be the  $k$ -th row of  $B$ , for  $1 \leq k \leq n$ . Clearly  $AB = \sum_k A_{*k} B_{k*}$ . (Note that  $A_{*k}$  is a column vector,  $B_{k*}$  a row vector, and  $A_{*k} B_{k*}$  is an  $n \times n$  matrix.) Let  $a_k$  be the number of non-zero elements in  $A_{*k}$  and let  $b_k$  be the number of non-zero elements in  $B_{k*}$ . (For brevity, we omit the bars over  $a_k$  and  $b_k$  used in Section 2.1. No confusion will arise here.) As explained in Section 2.1, we can naively compute  $AB$  using  $O(\sum_k a_k b_k)$  operations. If  $A$  and  $B$  each contain  $m$  non-zero elements, then  $\sum_k a_k b_k$  may be as high as  $mn$ . (See the example in Section 2.1.)

For any subset  $I \subseteq [n]$  let  $A_{*I}$  be the submatrix composed of the columns of  $A$  whose indices are in  $I$  and let  $B_{I*}$  be the submatrix composed of the rows of  $B$  whose indices are in  $I$ . If  $J = [n] - I$ , then we clearly have  $AB = A_{*I} B_{I*} + A_{*J} B_{J*}$ . Note that  $A_{*I} B_{I*}$  and  $A_{*J} B_{J*}$  are both *rectangular* matrix multiplications. Recall that  $M(n, \ell, n)$  is the cost of multiplying an  $n \times \ell$  matrix by an  $\ell \times n$  matrix using the fastest rectangular matrix multiplication algorithm.

Let  $\pi$  be a permutation for which  $a_{\pi(1)} b_{\pi(1)} \geq a_{\pi(2)} b_{\pi(2)} \geq \dots \geq a_{\pi(n)} b_{\pi(n)}$ . A permutation  $\pi$  satisfying this requirement can be easily found in  $O(n)$  time using radix sort. The algorithm chooses a value  $1 \leq \ell \leq n$ , in a way that will be specified shortly, and sets  $I = \{\pi(1), \dots, \pi(\ell)\}$  and  $J = \{\pi(\ell+1), \dots, \pi(n)\}$ . The product  $A_{*I} B_{I*}$  is then computed using the fastest available rectangular matrix multiplication algorithm, using  $M(n, \ell, n)$  operations, while the product  $A_{*J} B_{J*}$  is computed naively using  $O(\sum_{k>\ell} a_{\pi(k)} b_{\pi(k)})$  operations. The two matrices  $A_{*I} B_{I*}$  and  $A_{*J} B_{J*}$  are added using  $O(n^2)$  operations. We naturally choose the value  $\ell$  that minimizes  $M(n, \ell, n) + \sum_{k>\ell} a_{\pi(k)} b_{\pi(k)}$ . (This can easily be done in  $O(n)$  time by simply checking all possible values.) The resulting algorithm, which we call *SMP* (Sparse Matrix Multiplication), is given in Figure 1. We now claim:

**Theorem 4.** *Algorithm  $SMP(A, B)$  computes the product of two  $n \times n$  matrices over a ring  $R$ , with  $m_1$  and  $m_2$  non-zero elements respectively, using at most*

$$O(\min\{ (m_1 m_2)^{\frac{\beta}{\beta+1}} n^{\frac{2-\alpha\beta}{\beta+1}+o(1)} + n^{2+o(1)}, m_1 n, m_2 n, n^{\omega+o(1)} \})$$

*ring operations.*

When  $m_1 = m_2 = m$ , the first term in the bound above is  $m^{\frac{2\beta}{\beta+1}} n^{\frac{2-\alpha\beta}{\beta+1}+o(1)}$ . It is easy to check that for  $m = O(n^{1+\frac{\alpha}{2}})$ , the number of operations performed by the algorithm is only  $n^{2+o(1)}$ . It is also not difficult to check that for  $m = O(n^{\frac{\omega+1}{2}-\epsilon})$ , for any  $\epsilon > 0$ , the algorithm performs only  $o(n^\omega)$  operations. Using the currently best available bounds on  $\omega, \alpha$  and  $\beta$ , namely  $\omega \simeq 2.376$ ,  $\alpha \simeq 0.294$ , and  $\beta \simeq 0.536$ , we get that the number of operations performed by the algorithm is at most  $O(m^{0.7} n^{1.2} + n^{2+o(1)})$ , justifying the claims made in the abstract and the introduction.

The proof of Theorem 4 relies on the following simple lemma:

**Algorithm**  $SMP(A, B)$ **Input:** Two  $n \times n$  matrices  $A$  and  $B$ .**Output:** The product  $AB$ .

1. Let  $a_k$  be the number of non-zero elements in  $A_{*k}$ , for  $1 \leq k \leq n$ .
2. Let  $b_k$  be the number of non-zero elements in  $B_{k*}$ , for  $1 \leq k \leq n$ .
3. Let  $\pi$  be a permutation for which  $a_{\pi(1)}b_{\pi(1)} \geq \dots \geq a_{\pi(n)}b_{\pi(n)}$ .
4. Find an  $0 \leq \ell \leq n$  that minimizes  $M(n, \ell, n) + \sum_{k>\ell} a_{\pi(k)}b_{\pi(k)}$ .
5. Let  $I = \{\pi(1), \dots, \pi(\ell)\}$  and  $J = \{\pi(\ell+1), \dots, \pi(n)\}$ .
6. Compute  $C_1 \leftarrow A_{*I}B_{I*}$  using a fast dense rectangular matrix multiplication algorithm.
7. Compute  $C_2 \leftarrow A_{*J}B_{J*}$  using the naive sparse matrix multiplication algorithm.
8. Output  $C_1 + C_2$ .

**Fig. 1.** The new fast sparse matrix multiplication algorithm.

**Lemma 1.** For any  $1 \leq \ell < n$  we have  $\sum_{k>\ell} a_{\pi(k)}b_{\pi(k)} \leq \frac{m_1 m_2}{\ell}$ .

*Proof.* Assume, without loss of generality, that  $a_1 \geq a_2 \geq \dots \geq a_n$ . Let  $1 \leq \ell < n$ . We then clearly have  $\sum_{k>\ell} a_{\pi(k)}b_{\pi(k)} \leq \sum_{k>\ell} a_k b_k$ . Also  $\ell a_{\ell+1} \leq \sum_{k \leq \ell} a_k \leq m_1$ . Thus  $a_{\ell+1} \leq m_1/\ell$ . Putting this together we get

$$\sum_{k>\ell} a_{\pi(k)}b_{\pi(k)} \leq \sum_{k>\ell} a_k b_k \leq a_{\ell+1} \sum_{k>\ell} b_k \leq \frac{m_1}{\ell} m_2 = \frac{m_1 m_2}{\ell}. \quad \square$$

We are now ready for the proof of Theorem 4.

*Proof. (of Theorem 4)* We first examine the two extreme possible choices of  $\ell$ . When  $\ell = 0$ , the naive matrix multiplication algorithm is used. When  $\ell = n$ , a fast dense square matrix multiplication algorithm is used. As the algorithm chooses the value of  $\ell$  that minimized the cost, it is clear that the number of operations performed by the algorithm is  $O(\min\{m_1 n, m_2 n, n^{\omega+o(1)}\})$ .

All that remains, therefore, is to show that the number of operations performed by the algorithm is also  $O((m_1 m_2)^{\frac{\beta}{\beta+1}} n^{\frac{2-\alpha\beta}{\beta+1}+o(1)} + n^{2+o(1)})$ . If  $m_1 m_2 \leq n^{2+\alpha}$ , let  $\ell = m_1 m_2 / n^2$ . As  $\ell \leq n^\alpha$ , we have,

$$M(n, \ell, n) + \sum_{k>\ell} a_{\pi(k)}b_{\pi(k)} \leq n^{2+o(1)} + \frac{m_1 m_2}{\ell} = n^{2+o(1)}.$$

If  $m_1 m_2 \geq n^{2+\alpha}$ , let  $\ell = (m_1 m_2)^{\frac{1}{\beta+1}} n^{\frac{\alpha\beta-2}{\beta+1}}$ . It is easy to verify that  $\ell \geq n^\alpha$ , and therefore

$$M(n, \ell, n) = n^{2-\alpha\beta+o(1)} \ell^\beta = (m_1 m_2)^{\frac{\beta}{\beta+1}} n^{\frac{2-\alpha\beta}{\beta+1}+o(1)},$$

$$\sum_{k>\ell} a_{\pi(k)} b_{\pi(k)} \leq \frac{m_1 m_2}{\ell} = (m_1 m_2)^{1-\frac{1}{\beta+1}} n^{\frac{2-\alpha\beta}{\beta+1}} = (m_1 m_2)^{\frac{\beta}{\beta+1}} n^{\frac{2-\alpha\beta}{\beta+1}}.$$

Thus,  $M(n, \ell, n) + \sum_{k>\ell} a_{\pi(k)} b_{\pi(k)} = (m_1 m_2)^{\frac{\beta}{\beta+1}} n^{\frac{2-\alpha\beta}{\beta+1} + o(1)}$ . As the algorithm chooses the value of  $\ell$  that minimizes the number of operations, this completes the proof of the theorem.  $\square$

## 4 Multiplying Three or More Matrices

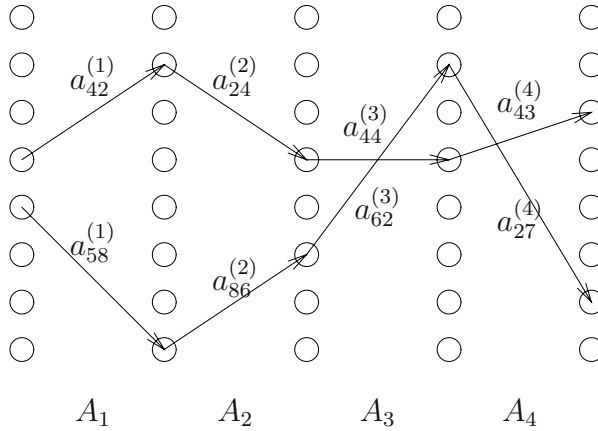
In this section we extend the results of the previous section to the product of three or more matrices. Let  $A_1, A_2, \dots, A_k$  be  $n \times n$  matrices, and let  $m_r$ , for  $1 \leq r \leq k$  be the number of non-zero elements in  $A_r$ . Let  $B = A_1 A_2 \cdots A_k$  be the product of the  $k$  matrices. As the product of two sparse matrices is not necessarily sparse, we cannot use the algorithm of the previous section directly to efficiently compute the product of more than two sparse matrices. Nevertheless, we show that the algorithm of the previous section can be generalized to efficiently handle the product of more than two matrices.

Let  $A_r = (a_{ij}^{(r)})$ , for  $1 \leq r \leq k$ , and  $B = A_1 A_2 \cdots A_k = (b_{ij})$ . It follows easily from the definition of matrix multiplication that

$$b_{ij} = \sum_{r_1, r_2, \dots, r_{k-1}} a_{i, r_1}^{(1)} a_{r_1, r_2}^{(2)} \cdots a_{r_{k-2}, r_{k-1}}^{(k-1)} a_{r_{k-1}, j}^{(k)}. \quad (1)$$

It is convenient to interpret the computation of  $b_{ij}$  as the summation over paths in a layered graph, as shown (for the case  $k = 4$ ) in Figure 2. More precisely, the layered graph corresponding to the product  $A_1 A_2 \cdots A_k$  is composed of  $k + 1$  layers  $V_1, V_2, \dots, V_{k+1}$ . Each layer  $V_r$ , where  $1 \leq r \leq k + 1$  is composed of  $n$  vertices  $v_{r,i}$ , for  $1 \leq i \leq n$ . For each non-zero element  $a_{ij}^{(r)}$  in  $A_r$ , there is an edge  $v_{r,i} \rightarrow v_{r+1,j}$  in the graph labelled by the element  $a_{ij}^{(r)}$ . The element  $b_{ij}$  of the product is then the sum over all directed paths in the graph from  $v_{1,i}$  to  $v_{k+1,j}$  of the product of the elements labelling the edges of the path.

Algorithm *SCMP* (Sparse Chain Matrix Multiplication) given in Figure 3 is a generalization of the variant of algorithm *SMP* given in the previous section for the product of two matrices. The algorithm starts by setting  $\ell$  to  $(\prod_{r=1}^k m_r)^{\frac{1}{k-1+\beta}} n^{\frac{\alpha\beta-2}{k-1+\beta}}$ , where  $m_r$  is the number of non-zero entries in  $A_r$ , for  $1 \leq r \leq k$ . (Note that when  $k = 2$ , we have  $\ell = (m_1 m_2)^{\frac{1}{\beta+1}} n^{\frac{\alpha\beta-2}{\beta+1}}$ , as in the proof of Theorem 4.) Next, the algorithm lets  $I_r$  be the set of indices of the  $\ell$  rows of  $A_r$  with the largest number of non-zero elements, ties broken arbitrarily, for  $2 \leq r \leq k$ . It also lets  $J_r = [n] - I_r$  be the set of indices of the  $n - \ell$  rows of  $A_r$  with the smallest number of non-zero elements. The rows of  $A_r$  with indices in  $I_r$  are said to be the *heavy* rows of  $A_r$ , while the rows of  $A_r$  with indices in  $J_r$  are said to be *light* rows. The algorithm is then ready to do some calculations. For every  $1 \leq r \leq k$ , it computes  $P_r \leftarrow (A_1)_{*J_2} (A_2)_{J_2 J_3} \cdots (A_r)_{J_r *}$ . This is done by enumerating all the corresponding paths in the layered graph corresponding to



**Fig. 2.** A layered graph corresponding to the product  $A_1 A_2 A_3 A_4$ .

the product. The matrix  $P_r$  is an  $n \times n$  matrix that gives the contribution of the *light* paths, i.e., paths that do not use elements from *heavy* rows of  $A_2, \dots, A_r$ , to the prefix product  $A_1 A_2 \cdots A_r$ . Next, the algorithm computes the suffix products  $S_r \leftarrow A_r \cdots A_k$ , for  $2 \leq r \leq k$ , using recursive calls to the algorithm. The cost of these recursive calls, as we shall see, will be overwhelmed by the other operations performed by the algorithm. The crucial step of the algorithm is the computation of  $B_r \leftarrow (P_{r-1})_{I_r*} (S_r)_{I_r*}$ , for  $2 \leq r \leq k$ , using the fastest available rectangular matrix multiplication algorithm. The algorithm then computes and outputs the matrix  $(\sum_{r=2}^k B_r) + P_k$ .

**Theorem 5.** Let  $A_1, A_2, \dots, A_k$  be  $n \times n$  matrices each with  $m_1, m_2, \dots, m_k$  non-zero elements, respectively, where  $k \geq 2$  is a constant. Then, algorithm  $\text{SCMP}(A_1, A_2, \dots, A_k)$  correctly computes the product  $A_1 A_2 \cdots A_k$  using

$$O \left( \left( \prod_{r=1}^k m_r \right)^{\frac{\beta}{k-1+\beta}} n^{\frac{(2-\alpha\beta)(k-1)}{k-1+\beta} + o(1)} + n^{2+o(1)} \right)$$

algebraic operations.

*Proof.* It is easy to see that the outdegree of a vertex  $v_{r,j}$  is the number of non-zero elements in the  $j$ -th row of  $A_r$ . We say that a vertex  $v_{r,j}$  is *heavy* if  $j \in I_r$ , and *light* otherwise. (Note that vertices of  $V_{k+1}$  are not classified as light or heavy. The classification of  $V_1$  vertices is not used below.) A path in the layered graph is said to be *light* if all its *intermediate* vertices are light, and *heavy* if at least one of its *intermediate* vertices is heavy.

Let  $a_{i,s_1}^{(1)} a_{s_1,s_2}^{(2)} \cdots a_{s_{k-2},s_{k-1}}^{(k-1)} a_{s_{k-1},j}^{(k)}$  be one of the terms appearing in the sum of  $b_{ij}$  given in (1). To prove the correctness of the algorithm we show that this term appears in exactly one of the matrices  $B_2, \dots, B_k$  and  $P_k$  which are added up to

**Algorithm**  $SCMP(A_1, A_2, \dots, A_k)$

**Input:**  $n \times n$  matrices  $A_1, A_2, \dots, A_k$ .

**Output:** The product  $A_1 A_2 \cdots A_k$ .

1. Let  $\ell = (\prod_{r=1}^k m_r)^{\frac{1}{k-1+\beta}} n^{\frac{\alpha\beta-2}{k-1+\beta}}$ .
2. Let  $I_r$  be the set of indices of  $\ell$  rows of  $A_r$  with the largest number of non-zero elements, and let  $J_r = [n] - I_r$ , for  $2 \leq r \leq k$ .
3. Compute  $P_r \leftarrow (A_1)_{*J_2} (A_2)_{J_2 J_3} \cdots (A_r)_{J_r *}$  by enumerating all corresponding paths, for  $1 \leq r \leq k$ .
4. Compute  $S_r \leftarrow A_r \cdots A_k$ , for  $2 \leq r \leq k$ , using recursive calls to the algorithm.
5. Compute  $B_r \leftarrow (P_{r-1})_{*I_r} (S_r)_{I_r *}$  using the fastest available rectangular matrix multiplication algorithm, for  $2 \leq r \leq k$ .
6. Output  $(\sum_{r=2}^k B_r) + P_k$ .

**Fig. 3.** Computing the product of several sparse matrices.

produce the matrix returned by the algorithm. Indeed, if the path corresponding to the term is light, then the term appears in  $P_k$ . Otherwise, let  $v_{r,j_r}$  be the first heavy vertex appearing on the path. The term then appears in  $B_r$  and in no other product. This completes the correctness proof.

We next consider the complexity of the algorithm. As mentioned, the outdegree of a vertex  $v_{r,j}$  is equal to the number of non-zero elements in the  $j$ -th row of  $A_r$ . The total number of non-zero elements in  $A_r$  is  $m_r$ . Let  $d_r$  be the maximum outdegree of a light vertex of  $V_r$ . The outdegree of every heavy vertex of  $V_r$  is then at least  $d_r$ . As there are  $\ell$  heavy vertices, it follows that  $d_r \leq m_r/\ell$ , for  $2 \leq r \leq k$ .

The most time consuming operations performed by the algorithm are the computations of

$$P_k \leftarrow (A_1)_{*J_2} (A_2)_{J_2 J_3} \cdots (A_k)_{J_k *}$$

by explicitly going over all light paths in the layered graph, and the  $k-1$  rectangular products

$$B_r \leftarrow (P_{r-1})_{*I_r} (S_r)_{I_r *} \quad , \quad \text{for } 2 \leq r \leq k .$$

The number of light paths in the graph is at most  $m_1 \cdot d_2 d_3 \cdots d_k$ . Using the bounds we obtained on the  $d_r$ 's, and the choice of  $\ell$  we get that the number of light paths is at most

$$\begin{aligned} m_1 \cdot d_2 d_3 \cdots d_k &\leq (\prod_{r=1}^k m_r) / \ell^{k-1} \\ &\leq (\prod_{r=1}^k m_r) \left[ (\prod_{r=1}^k m_r)^{-\frac{k-1}{k-1+\beta}} n^{\frac{(2-\alpha\beta)(k-1)}{k-1+\beta}} \right] = (\prod_{r=1}^k m_r)^{\frac{\beta}{k-1+\beta}} n^{\frac{(2-\alpha\beta)(k-1)}{k-1+\beta}} . \end{aligned}$$

Thus, the time taken to compute  $P_k$  is  $O((\prod_{r=1}^k m_r)^{\frac{\beta}{k-1+\beta}} n^{\frac{(2-\alpha\beta)(k-1)}{k-1+\beta}})$ . (Computing the product of the elements along a path requires  $k$  operations, but we consider  $k$  to be a constant.)

As  $|I_r| = \ell$ , for  $2 \leq r \leq k$ , the product  $(P_{r-1})_{*I_r}(S_r)_{I_r*}$  is the product of an  $n \times \ell$  matrix by an  $\ell \times n$  matrix whose cost is  $M(n, \ell, n)$ . Using Corollary 1 and the choice of  $\ell$  made by the algorithm we get that

$$\begin{aligned} M(n, \ell, n) &= n^{2-\alpha\beta+o(1)} \ell^\beta + n^{2+o(1)} \\ &= n^{2-\alpha\beta+o(1)} \left[ (\prod_{r=1}^k m_r)^{\frac{\beta}{k-1+\beta}} n^{\frac{(\alpha\beta-2)\beta}{k-1+\beta}} \right] + n^{2+o(1)} \\ &= (\prod_{r=1}^k m_r)^{\frac{\beta}{k-1+\beta}} n^{(2-\alpha\beta)(1-\frac{\beta}{k-1+\beta})+o(1)} + n^{2+o(1)} \\ &= (\prod_{r=1}^k m_r)^{\frac{\beta}{k-1+\beta}} n^{\frac{(2-\alpha\beta)(k-1)}{k-1+\beta}+o(1)} + n^{2+o(1)}. \end{aligned}$$

Finally, it is easy to see that the cost of computing the suffix products  $S_r \leftarrow A_r \cdots A_k$ , for  $2 \leq r \leq k$ , using recursive calls to the algorithm, is dominated by the cost of the other operations performed by the algorithm. (Recall again that  $k$  is a constant.) This completes the proof of the theorem.  $\square$

There are two alternatives to the use of algorithm *SCMP* for computing the product  $A_1 A_2 \cdots A_k$ . The first is to ignore the sparsity of the matrices and multiply the matrices in  $O(n^\omega)$  time. The second is to multiply the matrices, one by one, using the naive algorithm. As the naive algorithm uses at most  $O(mn)$  operations when *one* of the matrices contains only  $m$  non-zero elements, the total number of operations in this case will be  $O((\sum_{i=r}^k m_r)n)$ . For simplicity, let us consider the case in which each one of the matrices  $A_1, \dots, A_k$  contains  $m$  non-zero elements. A simple calculation then shows that *SCMP* is faster than the fast dense matrix multiplication algorithm for

$$m \leq n^{\frac{k-1+\omega}{k}},$$

and that it is faster than the naive matrix multiplication algorithm for

$$m \geq \max\left\{n^{\frac{k-\beta(1+(k-1)\alpha)-1}{(k-1)(1-\beta)}}, n^{1+o(1)}\right\}.$$

For  $k = 2$ , these bounds coincide with the bounds obtained in Section 3. For  $k = 3$ , with the best available bounds on  $\omega, \alpha$  and  $\beta$ , we get that *SCMP* is the fastest algorithm when  $n^{1.24} \leq m \leq n^{1.45}$ . For smaller values of  $m$  the naive algorithm is the fastest, while for larger values of  $m$  the fast dense algorithm is the fastest. Sadly, for  $k \geq 4$ , with the current values of  $\omega, \alpha$  and  $\beta$ , the new algorithm never improves on both the naive and the dense algorithms. But, this may change if improved bounds on  $\omega$ , and especially on  $\alpha$ , are obtained.

## 5 Concluding Remarks and Open Problems

We obtained an improved algorithm for the multiplication of two sparse matrices. The algorithm does not rely on any specific structure of the matrices to be

multiplies, just on the fact that they are sparse. The algorithm essentially partitions the matrices to be multiplied into a dense part and a sparse part and uses a fast algebraic algorithm to multiply the dense parts, and the naive algorithm to multiply the sparse parts. We also discussed the possibility of extending the ideas to the product of  $k \geq 3$  matrices. For  $k = 3$  we obtained some improved results. The new algorithms were presented for square matrices. It is not difficult, however, to extend them to work on rectangular matrices.

The most interesting open problem is whether it is possible to speed up the running time of other operations on sparse matrices. In particular, is it possible to compute the transitive closure of a directed  $n$ -vertex  $m$ -edge graph in  $o(mn)$  time? Is there an  $O(m^{1-\epsilon}n^{1+\epsilon})$  algorithm for the problem, for some  $\epsilon > 0$ ?

## References

1. N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42:844–856, 1995.
2. N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.
3. P. Bürgisser, M. Clausen, and M.A. Shokrollahi. *Algebraic complexity theory*. Springer-Verlag, 1997.
4. T. Chan. Dynamic subgraph connectivity with geometric applications. In *Proc. of 34th STOC*, pages 7–13, 2002.
5. J. Cheriyan. Randomized  $\tilde{O}(M(|V|))$  algorithms for problems in matching theory. *SIAM Journal on Computing*, 26:1635–1655, 1997.
6. H. Cohn and C. Umans. A group-theoretic approach to fast matrix multiplication. In *Proc. of 44th FOCS*, pages 438–449, 2003.
7. D. Coppersmith. Rectangular matrix multiplication revisited. *Journal of Complexity*, 13:42–49, 1997.
8. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.
9. C. Demetrescu and G.F. Italiano. Fully dynamic transitive closure: Breaking through the  $O(n^2)$  barrier. In *Proceedings of FOCS'00*, pages 381–389, 2000.
10. F. Eisenbrand and F. Grandoni. Detecting directed 4-cycles still faster. *Information Processing Letters*, 87(1):13–15, 2003.
11. F.G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, 1978.
12. X. Huang and V.Y. Pan. Fast rectangular matrix multiplications and applications. *Journal of Complexity*, 14:257–299, 1998.
13. D. Kratsch and J. Spinrad. Between  $O(nm)$  and  $O(n^\alpha)$ . In *Proc. of 14th SODA*, pages 709–716, 2003.
14. K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–113, 1987.
15. J. Nešetřil and S. Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.
16. V. Pan. *How to multiply matrices faster*. Lecture notes in computer science, volume 179. Springer-Verlag, 1985.

17. M.O. Rabin and V.V. Vazirani. Maximum matchings in general graphs through randomization. *Journal of Algorithms*, 10:557–567, 1989.
18. R. Raz. On the complexity of matrix product. *SIAM Journal on Computing*, 32:1356–1369, 2003.
19. L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proceedings of FOCS'02*, pages 679–689, 2002.
20. R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51:400–403, 1995.
21. A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. of 40th FOCS*, pages 605–614, 1999.
22. A. Shpilka. Lower bounds for matrix product. *SIAM Journal on Computing*, 32:1185–1200, 2003.
23. V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.
24. R. Yuster and U. Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proc. of 15th SODA*, pages 247–253, 2004.
25. U. Zwick. All-pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49:289–317, 2002.



# An Experimental Study of Random Knapsack Problems<sup>\*</sup>

Rene Beier<sup>1</sup> and Berthold Vöcking<sup>2</sup>

<sup>1</sup> Max-Planck-Institut für Informatik, Saarbrücken, Germany  
rbeier@mpi-sb.mpg.de

<sup>2</sup> Fachbereich Informatik, Universität Dortmund, Germany  
voecking@ls2.cs.uni-dortmund.de

**Abstract.** The size of the Pareto curve for the bicriteria version of the knapsack problem is polynomial on average. This has been shown for various random input distributions. We experimentally investigate the number of Pareto optimal knapsack fillings. Our experiments suggest that the theoretically proven upper bound of  $O(n^3)$  for uniform instances and  $O(\phi\mu n^4)$  for general probability distributions is not tight. Instead we conjecture an upper bound of  $O(\phi\mu n^2)$  matching a lower bound for adversarial weights.

In the second part we study advanced algorithmic techniques for the knapsack problem. We combine several ideas that have been used in theoretical studies to bound the average-case complexity of the knapsack problem. The concepts used are simple and have been known since at least 20 years, but apparently have not been used together. The result is a very competitive code that outperforms the best known implementation *Combo* by orders of magnitude also for harder random knapsack instances.

## 1 Introduction

Given  $n$  items with positive weights  $w_1, \dots, w_n$  and profits  $p_1, \dots, p_n$  and a knapsack capacity  $c$ , the knapsack problem asks for a subset  $S \subseteq [n] := \{1, 2, \dots, n\}$  such that  $\sum_{i \in S} w_i \leq c$  and  $\sum_{i \in S} p_i$  is maximized. Starting with the pioneering work of Dantzig [4], the problem has been studied extensively in practice (e.g. [13, 14, 17]) as well as in theory (e.g. [12, 9, 5, 2, 3]). For a comprehensive treatment we refer to the recent monograph by Kellerer, Pferschy and Pisinger [11] which is entirely devoted to the knapsack problem and its variations. Despite the NP-hardness of the problem [8], several large scale instances can be solved to optimality very efficiently. In particular, randomly generated instances seem to be quite easy to solve. As most exact algorithms rely on a partial enumeration of knapsack fillings, pruning the search space is an important issue for efficiency. A standard approach is the domination concept [18]. Consider the bicriteria version of the knapsack problem. Instead of a strict bound on the weight of knapsack fillings, assume that the two objectives (weight and profit) are to be optimized simultaneously, i.e., we aim at minimizing weight and maximizing profit. The set of solutions is then  $\mathcal{S} = 2^{[n]}$ .

---

<sup>\*</sup> This work was supported in part by DFG grant Vo889/1-1.

*Pareto optimal solutions.* A solution dominates another solution if it is better or equally good in all objectives and strictly better in at least one objective. A solution  $S$  is called *Pareto optimal* or *undominated*, if there is no other solution that dominates  $S$ . Let  $w(S) = \sum_{i \in S} w_i$  and  $p(S) = \sum_{i \in S} p_i$  denote the weight and profit of solution  $S \in \mathcal{S}$ . The set of pairs  $\{(w(S), p(S)) \mid S \in \mathcal{S} \text{ is Pareto optimal}\}$  is called *Pareto points* or *Pareto curve* of  $\mathcal{S}$ . These definitions translate to the more general framework of multiobjective combinatorial optimization [6], where the notion of Pareto optimality plays an crucial roll. In particular, the problem of enumerating all Pareto optimal solutions is of major interest and has led to a long list of publications, compiled in [7]. For the knapsack problem, however, enumerating Pareto optimal solutions is easy, due to the simple combinatorial structure of the problem.

Observe that enumerating all Pareto optimal knapsack fillings essentially solves the knapsack problem as the most profitable solution that satisfies the capacity bound  $c$  is an optimal solution. The reason is that every feasible dominated solution cannot be better than the solution it is dominated by, which, by definition, is also feasible. Notice that the number of Pareto points and Pareto optimal solutions can differ heavily since many Pareto optimal solutions can be mapped to the same Pareto point. We will sometimes blur this distinction for the sake of a short presentation.

*The Nemhauser/Ullmann algorithm.* In 1969, Nemhauser and Ullmann [16] introduce the following elegant algorithm that computes a list of all Pareto points in an iterative manner. In the literature (e.g. [11]) the algorithm is sometimes referred as “Dynamic programming with lists”. For  $i \in [n]$ , let  $\mathcal{L}_i$  be the list of all Pareto points over the solution set  $\mathcal{S}_i = 2^{[i]}$ , i.e.,  $\mathcal{S}_i$  contains all subsets of the items  $1, \dots, i$ . Recall that each Pareto point is a (weight, profit) pair. The points in  $\mathcal{L}_i$  are assumed to be listed in increasing order of their weights. Clearly,  $\mathcal{L}_1 = \langle (0, 0), (w_1, p_1) \rangle$ . The list  $\mathcal{L}_{i+1}$  can be computed from  $\mathcal{L}_i$  as follows: Create a new ordered list  $\mathcal{L}'_i$  by duplicating  $\mathcal{L}_i$  and adding  $(w_{i+1}, p_{i+1})$  to each point. Now we merge the two lists into  $\mathcal{L}_{i+1}$  obeying the weight order of subsets. Finally, those solutions are removed from the list that are dominated by other solutions in the list. The list  $\mathcal{L}_i$  can be calculated from the list  $\mathcal{L}_{i-1}$  in time that is linear in the length of  $\mathcal{L}_{i-1}$ . Define  $q(i) = |\mathcal{L}_i|$ . Thus, the Nemhauser/Ullmann algorithm computes the Pareto curve in time  $O(\sum_{i=1}^n q(i-1))$ .

In order to extract the knapsack fillings corresponding to the computed Pareto points, we maintain a bit for each Pareto point in  $\mathcal{L}_i$  indicating whether or not item  $i$  is contained in the corresponding knapsack filling. The knapsack filling can then reconstructed recursively using the standard technique from dynamic programming (see e.g. [11]). In a recent theoretical study, the average number of Pareto points for the knapsack problem has been proved to be polynomial under various random input distributions.

**Theorem 1.** ([2]) Suppose the weights are arbitrary positive numbers. Let  $q$  denote the number of Pareto points over all  $n$  items and let  $T$  denote the running time of the Nemhauser/Ullmann algorithm.

- If profits are chosen according to the uniform distribution over  $[0, 1]$  then  $E[q] = O(n^3)$  and  $E[T] = O(n^4)$ .

- For every  $i \in [n]$ , let profit  $p_i$  be a non-negative random variable with density function  $f_i : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ . Suppose  $\mu = \max_{i \in [n]} (\mathbf{E}[p_i])$  and  $\phi = \max_{i \in [n]} (\sup_{x \in \mathbb{R}} f_i(x))$ . Then  $\mathbf{E}[q] = O(\phi \mu n^4)$  and  $\mathbf{E}[q] = O(\phi \mu n^5)$ .

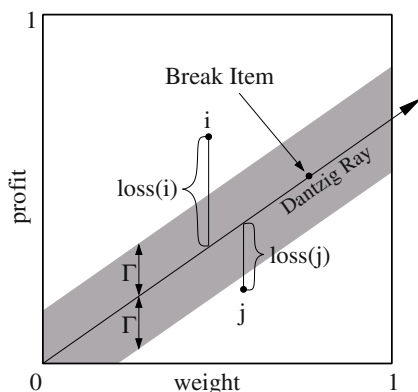
In fact, the proof technique can easily be adapted to show a more general theorem [1] which bounds the expected number of Pareto points over an arbitrary collection of subsets of  $n$  elements (items) and which holds for any optimization direction for the two objectives. This way, it applies to general bicriteria problems with linear objective functions for which the solution set of an instance can be described by a collection of subsets over a ground set. As an example, consider the bicriteria shortest path problem. The ground set is the set of edges and solutions are elementary  $s, t$ -paths, hence, subsets of edges. This generalized theorem has been applied to a simple algorithm enumerating all Pareto optimal  $s, t$ -paths and, thus, led to the first average-case analysis [1] for the constrained shortest path problem [15]. We believe that the generalized version of the theorem hold much potential of improving and simplifying probabilistic analyses for other combinatorial bicriteria problems and their variations.

An interesting question concerns the tightness of the upper bounds. There exists a lower bound of  $n^2/4$  for non-decreasing density functions for knapsack instances using exponentially growing weights [2]. This leaves a gap of order  $n$  for the uniform distribution and order  $n^2$  for general distributions. In the first part of this experimental study we investigate the number of Pareto points under various random input distributions. Even though the experiments are limited to knapsack instances and the generality of the upper bounds (adversarial weights, arbitrary probability distributions, different distributions for different items) hardly allows a qualitative confirmation of the bounds, we found some evidence that the lower bound is tight, which in turn opens the chance of improving the upper bounds.

In the second part we study the influence of various algorithmic concepts on the efficiency of knapsack algorithms. A combination of all considered ideas led to a very competitive code. Interestingly, all concepts used are long known (at least since 1984) but apparently have not been used together in an implementation.

The most influential algorithmic idea is the core concept. Core algorithms make use of the fact that the optimal integral solution is usually close to the optimal fractional one in the sense that only few items need to be exchanged in order to transform one into the other. Obtaining an optimal fractional solution is computationally very inexpensive. Based on this solution, a subset of items, called the core, is selected. The core problem is to find an optimal solution under the restriction that items outside the core are fixed to the value prescribed by the optimal fractional solution. The hope is that a small core is sufficient to obtain an optimal solution for the unrestricted problem. The core problem itself is again a knapsack problem with a different capacity bound and a subset of the original items. Intuitively, the core should contain those items for which it is hard to decide whether or not they are part of an optimal knapsack filling. The core concept leaves much space for variations and heuristics, most notably, the selection of the core items. A common approach selects items whose profit-to-weight ratio is closest to the profit-to-weight ratio of the break item [11]. We, however, follow the definition of Goldberg and Marchetti-Spaccamela [9], who introduce the notion of the loss of items.

**Fig. 1.** Items correspond to points in the  $xy$ -plane. The ray starting from the origin through the break item is called Dantzig ray. Items above the Dantzig ray are included in the break solution, items below the Dantzig ray are excluded. The loss of an item is the vertical distance to the Dantzig ray. The core region (shaded area) is a stripe around the ray. It grows dynamically until the vertical thickness has reached the size of the integrality gap  $\Gamma$ . Items outside the core are fixed to the value prescribed by the break solution.



Let us first describe the greedy algorithm for computing an optimal fractional solution. Starting with the empty knapsack, we add items one by one in order of non-increasing profit-to-weight ratio. The algorithm stops when it encounters the first item that would exceed the capacity bound  $c$ . This item is called *break item* and the computed knapsack filling, not including the *break item*, is called the *break solution*. Finally, we fill the residual capacity with an appropriate fraction  $f < 1$  of the break item. Let  $\bar{X} = (\bar{x}_1, \dots, \bar{x}_n)$  be the binary solution vector of this solution (or any other optimal fractional solution). For any feasible integer solution  $X = (x_1, \dots, x_n)$ , define  $ch(X) = \{i \in [n] : \bar{x}_i \neq x_i\}$ , i.e.,  $ch(X)$  is the set of items on which  $\bar{X}$  and  $X$  do not agree. When removing an item from  $\bar{X}$ , the freed capacity can be used to include other items which have profit-to-weight ratio at most that of the break item. A different profit-to-weight ratio of the exchanged items causes a loss in profit that can only be compensated by better utilizing the knapsack capacity. This loss can be quantified as follows:

**Definition 1.** Let  $r$  denote the profit-to-weight ratio of the break item. Define the loss of item  $i$  to be  $loss(i) = |p_i - r w_i|$ .

Goldberg and Marchetti-Spaccamela [9] proved that the difference in profit between any feasible integer solution  $X = (x_1, \dots, x_n)$  and the optimal fractional solution  $\bar{X}$  can be expressed as

$$p(\bar{X}) - p(X) = r \left( c - \sum_{i=1}^n x_i w_i \right) + \sum_{i \in ch(X)} loss(i) .$$

The first term on the right hand side corresponds to an unused capacity of the integer solution  $X$  whereas the second term is due to the accumulated loss of all items in  $ch(X)$ . Let  $X^*$  denote an arbitrary optimal integral solution. The integrality gap  $\Gamma = p(\bar{X}) - p(X^*)$  gives an upper bound for the accumulated loss of all items in  $ch(X^*)$ , and therefore, also for the loss of each individual item in  $ch(X^*)$ .

**Fact 1.** For any optimal integer solution  $X^*$  it holds  $\sum_{i \in ch(X^*)} loss(i) \leq \Gamma$ .

Thus, items with loss larger than  $\Gamma$  must agree in  $\bar{X}$  and  $X^*$ . Consequently, we define the core to consist of all elements with loss at most  $\Gamma$ . Since we do not know  $\Gamma$ , we

use a dynamic core, adding items one by one in order of increasing loss and compute a new optimal solution whenever we extend the core. We stop when the next item has loss larger than the current integrality gap, that is, the gap in profit between  $\bar{X}$  and the best integer solution found so far. Figure 1 illustrates the definitions of the core.

*Loss filter.* Goldberg and Marchetti-Spaccamela further exploit Fact 1. Observe that solving the core problem is equivalent to finding the set  $ch(X^*)$  for some optimal solution  $X^*$ . Let  $I_B$  denote the set of core items included in the break solution. We solve a core problem relative to the break solution, that is, including an item from  $I_B$  into a core solution corresponds to removing it from the break solution. Hence, in the core problem, items in  $I_B$  have negative weight and profit. The capacity of this core problem is  $c_B$ , the residual capacity of the break solution. The loss of a core solution  $X$  is  $\text{loss}(X) = \sum_{i=1}^n \text{loss}(i)x_i$ . According to Fact 1, solutions with loss larger than the integrality gap (or any upper bound, e.g., provided by the best integer solution found so far) can not be optimal and, furthermore, cannot be extended to an optimal solution by adding more core items. These solutions are filtered out in the enumeration process.

*Horowitz and Sahni decomposition.* Horowitz and Sahni [10] present a variation of the Nemhauser/Ullmann algorithm utilizing two lists of Pareto points. Assume we partition the set of items into  $I$  and  $I'$ . Let  $\mathcal{L}$  and  $\mathcal{L}'$  denote the sorted lists of Pareto points over  $I$  and  $I'$ , respectively. Each Pareto point over the complete item set  $I \cup I'$  can be constructed by combining two Pareto points from  $\mathcal{L}$  and  $\mathcal{L}'$  (or unifying the corresponding subsets of items). Instead of checking all pairs  $(p, q) \in \mathcal{L} \times \mathcal{L}'$ , it suffices to find for each  $p \in \mathcal{L}$  the most profitable  $q \in \mathcal{L}'$  such that the combination of  $p$  and  $q$  is still feasible. Since the lists are sorted, the  $p \in \mathcal{L}$  with the most profitable combination can be found by a single parallel scan of the two lists in opposite directions.

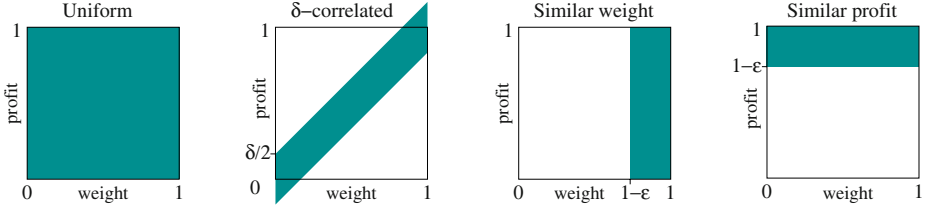
## 1.1 Experimental Setup

In order to avoid effects that are caused by the pseudopolynomial complexity of the algorithms we used a large integer domain of  $[0, 2^{30} - 1]$  for the weights and profits of items. The implementation uses 64 bit integer arithmetic (c++ type *long long*). All programs have been compiled under SunOS 5.9 using the GNU compiler g++, version 3.2.1 with option “-O4”. All experiments have been performed on a Sunfire<sup>®</sup> 15k with 176GB of main memory and 900MHz SparcIII+ CPUs (SPECint2000= 535 for one processor, according to [www.specbench.org](http://www.specbench.org)).

*Random input distributions.* Following the definition in [11],  $X \leftarrow \mathcal{U}[l, r]$  means that the random variable  $X$  is chosen independently uniformly at random from the interval  $[l, r]$ . We investigate the following input distributions:

**Uniform instances:**  $w \leftarrow \mathcal{U}[0, 1]$  and  $p \leftarrow \mathcal{U}[0, 1]$ .

**$\delta$ -correlated instances:**  $w \leftarrow \mathcal{U}[0, 1]$  and  $p \leftarrow w + r$  with  $r \leftarrow \mathcal{U}[-\delta/2, \delta/2]$  for a given  $0 < \delta \leq 1$ . Notice that  $\delta$ -correlated instances are a parameterized version of *weakly correlated instances* used in former studies (e.g. [11,13,17,14]). The latter correspond to  $\delta$ -correlated instances with  $\delta = 1/10$ . The corresponding density function of the profits is upper bounded by  $\phi = 1/\delta$ .



**Fig. 2.** Classes of test instances: Items are samples uniformly from the shaded area. To obtain integers, we scale each number by  $R$  and round to the next integer.

**Instances with similar weight:**  $w_i \leftarrow \mathcal{U}[(1-\epsilon), 1]$  and  $w_i \leftarrow \mathcal{U}[0, 1]$ .

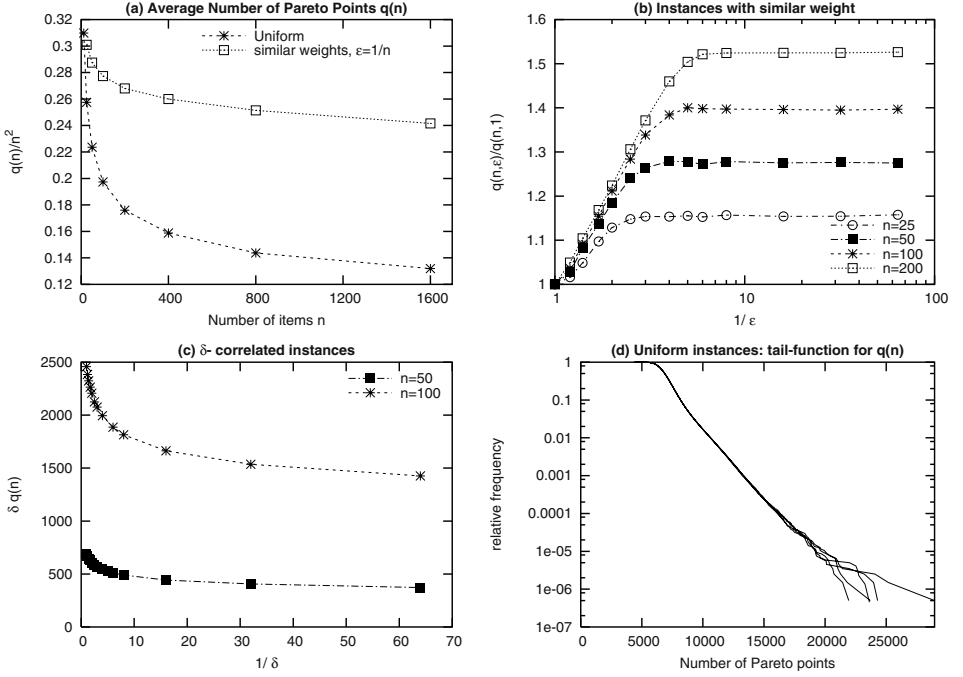
**Instances with similar profit:**  $w_i \leftarrow \mathcal{U}[0, 1]$  and  $w_i \leftarrow \mathcal{U}[(1-\epsilon), 1]$ .

In order to obtain integer weights and profits that can be processed by our algorithm, we scale all values by  $R$  and round to the next integer. In this study we used  $R = 2^{30} - 1$  for all experiments. These definitions are illustrated in Figure 2. Observe that for  $\delta$ -correlated instances some items can have negative profits. This affects a fraction  $\delta/8$  of all items on average. These items have no effect on the Pareto curve. Nevertheless they are processed by the Nemhauser/Ullmann algorithm. The reason to include these instances in our study is a theoretical analysis of a core algorithm [3] for this class of instances. The capacity  $c$  of the knapsack is set to  $\beta \cdot \sum_{i=1}^n w_i$ , with parameter  $\beta < 1$  specified for each experiment. If not indicated otherwise, each data point represents the average over  $T$  trials where  $T$  is a parameter specified for each experiment.

## 2 Number of Pareto Points

In the following,  $q(n)$  denotes the (average) number of Pareto points over  $n$  items computed by the Nemhauser/Ullmann algorithm.

First we tested uniform instances (Figure 3a). We observe that the ratio  $q(n)/n^2$  is decreasing in  $n$  indicating that  $q(n) = O(n^2)$ . As the decrease might be caused by lower order terms, the result could be tight. The next experiment for instances with similar weight gives some evidence for the contrary. Notice that by choosing  $\epsilon = 1$ , we obtain uniform instances. Figure 3b shows the dependence of  $\epsilon$  on average number of Pareto points for instances with similar weight. When decreasing  $\epsilon$  then  $q(n, \epsilon)$  first increases about linearly but stays constant when  $\epsilon$  falls below some value that depends on  $n$ . At this value, subsets of different cardinality are well separated in the Pareto curve, i.e., if  $\mathcal{P}_i$  denotes the set of Pareto optimal solutions with cardinality  $i$  then all knapsack fillings in  $\mathcal{P}_i$  have less weight (and less profit) than any solution in  $\mathcal{P}_{i+1}$ . For  $\epsilon = 1/n$ , subsets are obviously separated, but the effect occurs already for larger values of  $\epsilon$ . A similar separation of subsets was used in the proof for the lower bound of  $n^2/4$ . The ratio  $q(n, \epsilon)/q(n, 1)$  apparently converges to  $c \cdot \log(n)$  for  $\epsilon \rightarrow 0$  and some  $c \in \mathbb{R}_{>0}$ . In this case, the separation would cause a logarithmic increase of  $q(n, \epsilon)$  compared to uniform instances. As our data suggests that  $\mathbb{E}[q(n, \epsilon)] = O(n^2)$  for instances with similar weight using  $\epsilon = 1/n$  (see Figure 3a), the asymptotic behavior of  $q(n)$  for uniform instances

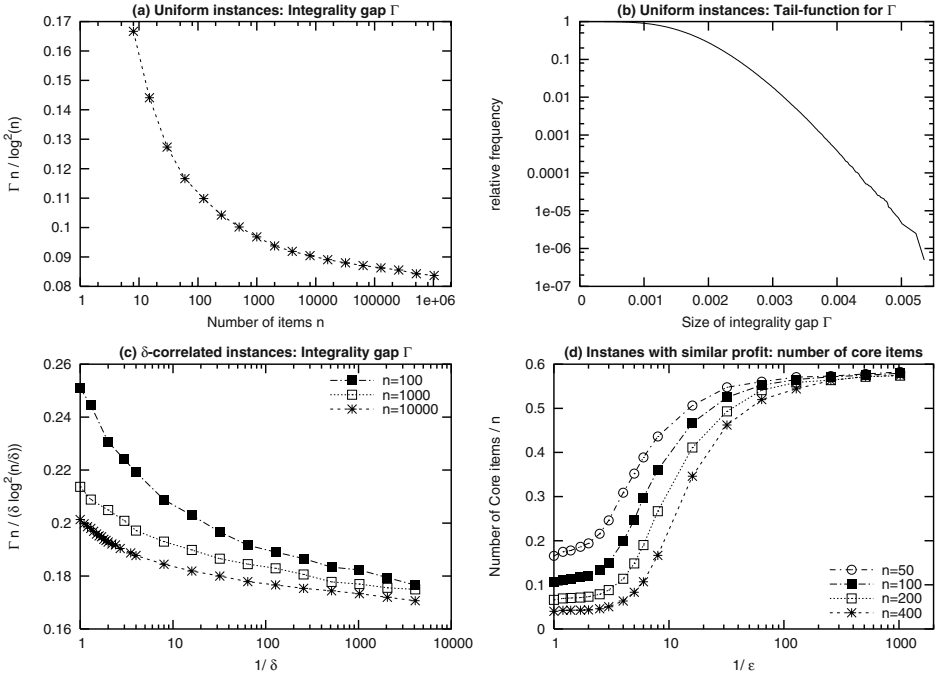


**Fig. 3.** Average number of Pareto points. **a)** Uniform instances and instances with similar weight using  $\epsilon = 1/n$ ,  $T \geq 2000$ : the ratio  $q(n)/n^2$  decreases in  $n$  indicating that  $q(n) = O(n^2)$ . **b)** Instances with similar weight,  $T = 10^4$ :  $q(n, \epsilon)/q(n, 1)$  converges to  $c \cdot \log(n)$  for  $\epsilon \rightarrow 0$  **c)**  $\delta$ -correlated instances,  $T = 10^4$ :  $\delta q(n)$  decreases in  $1/\delta$  indicating that  $q(n) = O(n^2/\delta)$ . **d)** Uniform instances: tail-functions for  $q(n)$ , graph shows the fraction of experiments with  $q(n) > x$ ,  $n=200$ . We run 5 identical experiments to examine the random deviations.

should be smaller than  $n^2$  by at least a logarithmic factor, i.e., there is evidence that  $\mathbf{E}[q(n)] = O(n^2/\log n)$ . The experimental data, however, does not allow any conclusion whether or not  $q(n, \epsilon) = \Theta(n^2)$  for  $\epsilon = 1/n$ . Notice that the results for instances with similar profit and instances with similar weight are the same. This is due to the fact that, for any given knapsack instance, exchanging the weight and profit of each item has no influence on the Pareto curve of that instance [1].

Next we consider  $\delta$ -correlated instances. As the density parameter for the profit distributions is  $\phi = 1/\delta$ , applying Theorem 1 yields a bound of  $O(n^4/\delta)$ . Figure 3a shows that  $\delta \cdot q(n, \delta)$  decreases in  $\phi = 1/\delta$ . Since uniform instances and  $\delta$ -correlated instances are quite similar for large  $\delta$ , the data proposes an asymptotic behavior of  $q(n, \delta) = O(n^2/\delta)$ . We conjecture that a separation of subsets with different cardinality would increase the average number of Pareto points by a logarithmic factor  $\log(n/\delta)$  and that  $O(n^2/\delta) = O(\phi n^2)$  is a tight bound for adversarial weights.

Finally we investigate the distribution of the random variable  $q(n)$  for uniform instances. Theorem 1 bounds the expected number of Pareto points. Applying the Markov inequality yields only a weak tail bound: for all  $\alpha \in \mathbb{R}_{>0}$ ,  $\Pr[q(n) > \alpha \mathbf{E}[q(n)]] \leq 1/\alpha$ .



**Fig. 4.** **a)** Average integrality gap  $\Gamma$  for uniform instances **b)** Tail function of  $\Gamma$  for uniform instances with  $n = 10000$  items: graph  $f(x)$  gives the fraction of experiments with  $\Gamma > x$ . **c)** Average integrality gap  $\Gamma$  for  $\delta$ -correlated instances. **d)** Random instances with similar profit: average number of core items as a function of  $\epsilon$ . We used  $T = 10000$ ,  $\beta = 0.4$  for all experiments.

Figure 3d shows a much faster decay of the probability (notice the log scale). For example, the probability that  $q(200)$  is larger than three times the average is about  $10^{-5}$  instead of  $1/3$ . The experiments suggest that the tail decreases faster than a polynomial  $1/\alpha^c$  but possibly slower than exponential  $1/c^\alpha$ .

In all our experiments (which we could not present here all) we have not found a distribution where  $q(n)$  was larger than  $\phi\mu n^2$  on average, where  $\phi$  and  $\mu$  denote the maximum density and the maximum expectation over the profit distributions of all items, respectively. This gives us some hope that the upper bound of  $O(\phi\mu n^4)$  can be improved.

### 3 Properties of Random Knapsack Instances

The probabilistic analyses in [9] and [3] are based (among other results) on the upper bound for the integrality gap  $\Gamma$ . Lueker [12] proved that  $\mathbf{E}[\Gamma] = O(\frac{\log^2 n}{n})$ , provided the weights and profits are drawn uniformly at random from  $[0, 1]$ . Lueker also gave a lower bound of  $\frac{1}{n}$  which was later improved to  $\Omega(\frac{\log^2 n}{n})$  by Goldberg and Marchetti-Spaccamela [9]. A complete proof of the last result, however, was never published. Our



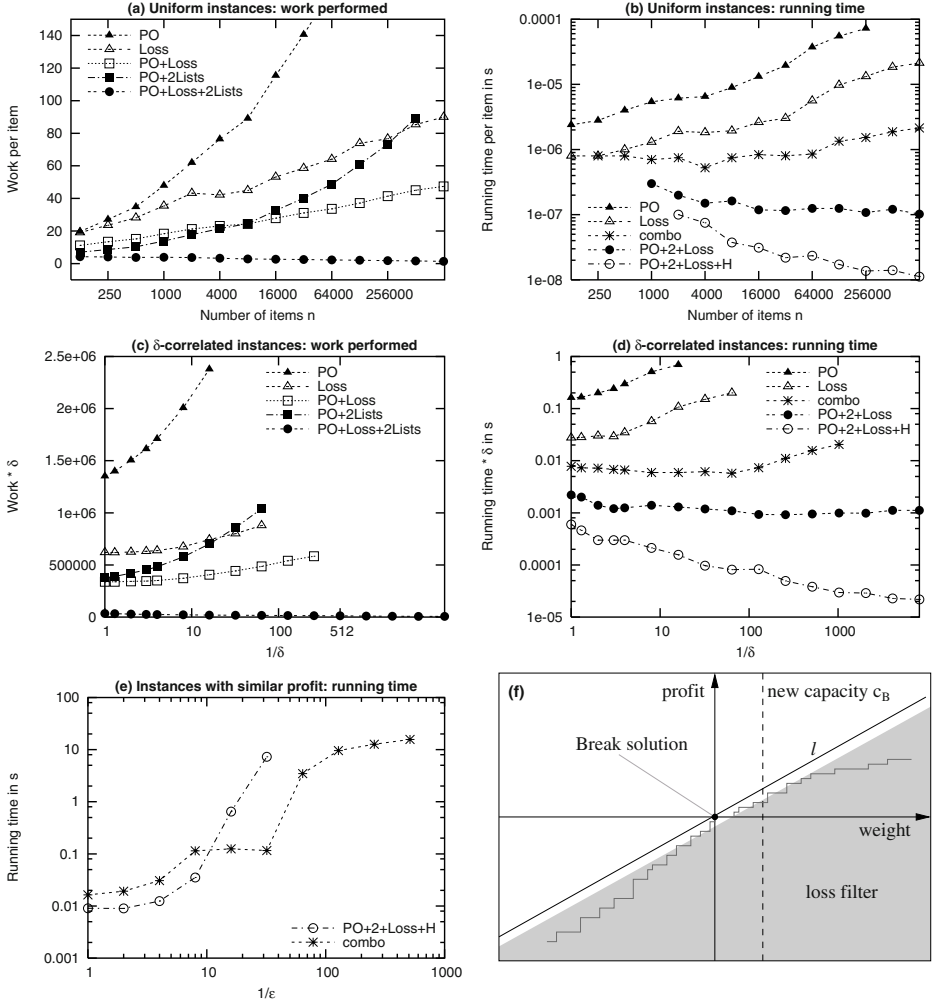
experiments are not sufficient to strongly support this lower bound. For an affirmative answer the curve in Figure 4a would need to converge to a constant  $c > 0$  (notice the log-scale for the number of items). Our experiments suggest that in this case the constant  $c$  would be at most  $1/12$ . The small constants involved also account for the moderate number of core items when using the core definition of Goldberg and Marchetti-Spaccamela. The average number of core items in our experiments was 20.2 for uniformly random instances with 1000 item and 67.5 for instances with  $10^6$  items.

Goldberg and Marchetti-Spaccamela show that Lueker's analysis for the upper bound of  $\Gamma$  can easily be generalized to obtain exponential tail bounds on  $\Gamma$ : there is constant  $c_0$  such that for every  $\alpha \leq \log^4 n$ ,  $\Pr[\Gamma \geq \alpha c_0 (\log n)^2 / n] \leq 2^{-\alpha}$ . Our experiment show an even more rapid decay (Figure 4b), also in terms of absolute values: We performed  $10^6$  experiments using  $n = 10000$  items. The average integrality gap was  $A = 1.695 \cdot 10^{-3}$ . The fraction of experiments in which  $\Gamma$  was larger than  $2A$  was  $4.5 \cdot 10^{-3}$ . Only in three experiments  $\Gamma$  was larger than  $3A$  and the largest  $\Gamma$  was about  $3.16A$ . Adapting Lueker's proof technique, a similar bound for the integrality gap of  $\delta$ -correlated instances was proven in [3]:  $\mathbf{E}[\Gamma] = O(\frac{\delta}{N} \log^2 \frac{N}{\delta})$ . Figure 4c comply with the theorem and, as in the uniform case, give a weak indication that the bound could be tight, as the curves show some tendency to converge to a constant  $c > 0$ .

Finally consider instances with similar profit. Instead of the integrality gap we investigated the number of core items. Recall that Goldberg and Marchetti-Spaccamela's definition of the core closely relate the two quantities. In particular, the core includes all items whose vertical distance to the Dantzig ray is at most  $\Gamma$ . Figure 4d shows that the core contains more than half of all items when  $\epsilon$  becomes small. The reason is that items close to the Dantzig ray have a very similar weight. If the remaining capacity of the knapsack is not close to the weight (or a multiple) of these items, its difficult to fill the knapsack close to its capacity. Therefore, items more distant to the Dantzig ray have to be used to fill the knapsack, but they exhibit a larger loss. As a consequence, the integrality gap is much larger on average and so is the number of core items.

## 4 Efficiency of Different Implementations

Let us first describe our implementation in more detail. First we compute the break solution  $\tilde{X}$  in linear time. Then we use a dynamic core, adding items in order if increasing loss. The capacity of the core problem is  $c - w(\tilde{X})$  and items included in  $\tilde{X}$  become negative, that is, we negate the weight and profit of those items. The core problem is solved using the Nemhauser/Ullmann algorithm, except for the variant that uses only the loss filter, as analyzed in [9]. The loss filter can easily be incorporated into the Nemhauser/Ullmann algorithm, as the loss of a Pareto point is determined by its weight and profit (see Figure 5f). Pareto points whose loss is larger than the "current" integrality gap are deleted from the list. Observe that all solutions dominated by such a filtered Pareto point have an even larger loss. Hence, domination concept and loss filter are "compatible". When using two lists (Horowitz and Sahni decomposition) we alternately extend the lists with the next item and check for a new optimal solution in each such iteration.



**Fig. 5.** Efficiency of different implementations: **a)** work and **b)** running time for uniform instances as a function of  $n$ . **c)** work and **d)** running time for  $\delta$ -correlated instances as a function of  $1/\delta$  using  $n = 10000$ . **e)** Uniform instances with similar profit using  $n = 10000$ . Each curve represents an implementation that uses a subset of the features as indicated: PO = filtering dominated solutions, Loss = loss filter, 2 = two lists, H = additional heuristics, combo = *Combo* code. Each data point represents the average over  $T = 1000$  experiments. We used  $\beta = 0.4$ . The given running times for uniform and  $\delta$ -correlated instances do not include the linear time preprocessing (the generation of the random instance, finding the optimal fractional solution, partitioning the set of items according to the loss of items) as it dominates the running time for easy instances.

**f)** Pareto curve of the core problem as a step function. Items included in the break solution are negated, as we solve a core problem relative to the Break solution. The slope of line  $l$  is the profit-to-weight ratio of the break item. Solutions in the shaded area (points whose vertical distance to  $l$  is larger than  $\Gamma$ ) are filtered out by the loss filter. Observe that if a Pareto point  $(w, p)$  is filtered out by the loss filter then all solutions dominated by  $(w, p)$  are in the shaded area as well.

*Additional heuristics.* Instead of alternately extending the two lists we first extend only one list, which grows exponentially at the beginning. Later we use the second list. The idea is that the first list is based on items with small loss such that the loss filter has hardly an effect. The items used in the second list are expected to have larger loss such that the loss filter keeps the list relatively small and the extension of the list with new items is fast. We use lazy evaluation, i.e., we do not scan the two lists in each iteration to find new optimal solutions as we expect the first list to be significantly larger than the second. Knowing the length of the lists, we balance the work for extending a list and scanning both lists. Finally, we delete Pareto points from the list that cannot be extended by forthcoming items as their loss is too large.

We investigate the influence of the different concepts (domination concept, loss filter, Horowitz and Sahni decomposition and additional heuristics) on the running time of our algorithm, under different random input distributions. We compare our implementation to *Combo*, a knapsack solver due to Pisinger [13]. *Combo* combines various advanced algorithmic techniques that have been independently developed to cope with specifically designed difficult test instances. For instance, it applies cardinality constraints generated from extended covers using Lagrangian relaxation, rudimentary divisibility tests and it pairs dynamic programming states with items outside the core. Our implementation, however, is rather simple but proves to be very efficient in reducing the number of enumerated knapsack filling for a given core. In particular, we add items strictly in order of increasing loss to the core. This can lead to a very large core, as shown in Figure 4d. We observed that whenever *Combo* was superior to our implementation, the core size of *Combo* was usually significantly smaller. So we consider our implementation not as a competing knapsack solver but rather as a study that points out possible improvements.

Due to technical difficulties with measuring small amounts of processing time and huge differences in the running times we employ, besides the running time, another measure, called work. Each basic step takes a unit work. This way, an iteration of the Nemhauser/Ullmann algorithm with input list  $\mathcal{L}$  costs  $2|\mathcal{L}|$  work units. Finding an optimal solution by scanning 2 lists  $\mathcal{L}_1$  and  $\mathcal{L}_2$  in parallel (Horowitz and Sahni decomposition) costs  $|\mathcal{L}_1| + |\mathcal{L}_2|$  work units.

Figure 5 shows the work and running time for different implementations. We expect the work as well as the running time to increase quasi-linear in  $n$  and  $1/\delta$  for uniform and  $\delta$ -correlated instances (see [3] for a theoretical result). For an easier comparison, we divided work and running time by  $n$  and  $1/\delta$ , accordingly. Notice that almost all scales are logarithmic.

In contrast to the theoretical analyses of core algorithms in [9] and [3], where the domination concept leads to a better running time bound than the loss filter, the latter is superior in all experiments. This remains true for large instances, even if the domination concept is complemented by the Horowitz/Sahni decomposition (2 lists of Pareto points). The *Combo* code is superior to all three variants. Interestingly, the combination of the 3 features (PO+Loss+2) clearly outperform *Combo* for uniform and  $\delta$ -correlated instances. In fact, it exhibits a sublinear running time for uniform instances (without linear time preprocessing). Using the described heuristics significantly improves the running times further. The average running time of *Combo* and our fastest implementation differs by a

factor of 190 for uniform instances with 1024000 items. For  $\delta$ -correlated instances with 10000 items and  $\delta = 1/1024$ , the factor is about 700.

The limitations of our implementation is illustrated in Figure 5e, which shows the running time (with preprocessing) for instances with similar profit. Besides the large core (Figure 4d), the large integrality gap makes the loss filter less efficient.

## References

1. R. Beier. *Probabilistic Analysis of Combinatorial Optimization Problems*. PhD thesis, Universität des Saarlandes, 2004. In preparation.
2. R. Beier and B. Vöcking. Random knapsack in expected polynomial time. In *Proc. 35th Annual ACM Symposium on Theory of Computing (STOC-2003)*, pages 232–241, San Diego, USA, 2003.
3. R. Beier and B. Vöcking. Probabilistic analysis of knapsack core algorithms. In *Proc. 15th Annual Symposium on Discrete Algorithms (SODA-2004)*, pages 461–470, New Orleans, USA, 2004.
4. G. B. Dantzig. Discrete variable extremum problems. *Operations Research*, 5:266–277, 1957.
5. M.E. Dyer and A. M. Frieze. Probabilistic analysis of the multidimensional knapsack problem. *Maths. of Operations Research*, 14(1):162–176, 1989.
6. M. Ehrgott. *Multicriteria optimization*. Springer Verlag, 2000.
7. M. Ehrgott and X. Gandibleux. *Multiple Criteria Optimization: State of the Art Annotated Bibliographic Surveys*, volume 52 of *International Series in Operations Research and Management Science*. Kluwer Academic Publishers, Boston, 2002.
8. M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman and Company, New York, 1979.
9. A. Goldberg and A. Marchetti-Spaccamela. On finding the exact solution to a zero-one knapsack problem. In *Proceedings of the 16th ACM Symposium on Theory of Computing (STOC)*, pages 359–368, 1984.
10. E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, April 1974.
11. H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.
12. G. S. Lueker. On the average difference between the solutions to linear and integer knapsack problems. *Applied Probability - Computer Science, the Interface, Birkhäuser*, 1:489–504, 1982.
13. S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424, 1999.
14. S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0-1 knapsack problem. *European Journal of Operational Research*, 123:325–332, 2000.
15. K. Mehlhorn and M. Ziegelmann. Resource constrained shortest paths. In *Proc. 8th Annual European Symp. (ESA-00)*, LNCS 1879, pages 326–337. Springer, 2000.
16. G. Nemhauser and Z. Ullmann. Discrete dynamic programming and capital allocation. *Management Science*, 15(9):494–505, 1969.
17. D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, DIKU, University of Copenhagen, 1995.
18. H.M. Weingartner and D.N. Ness. Methods for the solution of the multi-dimensional 0/1 knapsack problem. *Operations Research*, 15(1):83–103, 1967.

# Contraction and Treewidth Lower Bounds<sup>\*</sup>

Hans L. Bodlaender<sup>1</sup>, Arie M.C.A. Koster<sup>2</sup>, and Thomas Wolle<sup>1</sup>

<sup>1</sup> Institute of Information and Computing Sciences, Utrecht University  
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands  
{hansb,thomasw}@cs.uu.nl

<sup>2</sup> Konrad-Zuse-Zentrum für Informationstechnik Berlin  
Takustraße 7, D-14194 Berlin, Germany  
koster@zib.de

**Abstract.** Edge contraction is shown to be a useful mechanism to improve lower bound heuristics for treewidth. A successful lower bound for treewidth is the degeneracy: the maximum over all subgraphs of the minimum degree. The degeneracy is polynomial time computable. We introduce the notion of contraction degeneracy: the maximum over all graphs that can be obtained by contracting edges of the minimum degree. We show that the problem to compute the contraction degeneracy is *NP*-hard, but for fixed  $k$ , it is polynomial time decidable if a given graph  $G$  has contraction degeneracy at least  $k$ . Heuristics for computing the contraction degeneracy are proposed and experimentally evaluated. It is shown that these can lead to considerable improvements to the lower bound for treewidth. A similar study is made for the combination of contraction with Lucena's lower bound based on Maximum Cardinality Search [12].

## 1 Introduction

It is about two decades ago that the notion of treewidth and the equivalent notion of partial  $k$ -tree were introduced. Nowadays, these play an important role in many theoretic studies in graph theory and algorithms, but also their use for practical applications is growing, see, e.g.: [10,11]. A first step when solving problems on graphs of bounded treewidth is to compute a tree decomposition of (close to) optimal width, on which often a dynamic programming approach is applied. Such a dynamic programming algorithm typically has a running time that is exponential in the treewidth of the graph. Since determining the treewidth of a graph is an *NP*-hard problem, it is rather unlikely to find efficient algorithms for computing the treewidth. Therefore, we are interested in lower and upper bounds for the treewidth of a graph.

This paper focuses on lower bounds for the treewidth. Good lower bounds can serve to speed up branch and bound methods, inform us about the quality

---

<sup>\*</sup> This work was partially supported by the DFG research group “Algorithms, Structure, Randomness” (Grant number GR 883/9-3, GR 883/9-4), and partially by the Netherlands Organisation for Scientific Research NWO (project *Treewidth and Combinatorial Optimisation*).

of upper bound heuristics, and in some cases, tell us that we should not use tree decompositions to solve a problem on a certain instance. A large lower bound on the treewidth of a graph implies that we should not hope for efficient dynamic programming algorithms that use tree decompositions for this particular instance.

In this paper, we investigate the combination of contracting edges with two existing lower bounds for treewidth, namely the degeneracy (or MMD [9]) and the MCSLB (the Maximum Cardinality Search Lower Bound), introduced by Lucena [12]. The definitions of these lower bounds can be found in Section 2. Contracting an edge is replacing its two endpoints by a single vertex, which is adjacent to all vertices at least one of the two endpoints was adjacent to. Combining the notion of contraction with degeneracy gives the new notion of *contraction degeneracy* of a graph  $G$ : the maximum over all graphs that can be obtained by contracting edges of  $G$  of the minimum degree. It provides us with a new lower bound for the treewidth of graphs. While unfortunately, computing the contraction degeneracy of a graph is  $NP$ -complete (as is shown in Section 3.2), the fixed parameter cases are polynomial time solvable (see Section 3.3), and there are simple heuristics that provide us with good bounds on several instances taken from real life applications (see Section 5).

We experimented with three different heuristics for the contraction degeneracy, based on repeatedly contracting a vertex of minimum degree with (a) a neighbor of smallest degree, (b) a neighbor of largest degree, or (c) a neighbor that has as few as possible common neighbors. We see that the last one outperforms the other two, which can be explained by noting that such a contraction ‘destroys’ only few edges in the graph. Independently, the heuristic with contraction with neighbors of minimum degree has been proposed as a heuristic for lower bounds for treewidth by Gogate and Dechter [7] in their work on branch and bound algorithms for treewidth.

The lower bound provided by MCSLB is never smaller than the degeneracy, but can be larger [3]. The optimisation problem to find the largest possible bound of MCSLB for a graph obtained from  $G$  by contracting edges is also  $NP$ -hard, and polynomial time solvable for the fixed parameter case (Section 4). We also studied some heuristics for this bound. In our experiments, we have seen that typically, the bound by MCSLB is equal to the degeneracy or slightly larger. In both cases, often a large increase in the lower bound is obtained when we combine the method with contraction. See Section 5 for results of our computational experiments. They show that contraction is a very viable idea for obtaining treewidth lower bounds.

A different lower bound for treewidth was provided by Ramachandramurthi [13,14]. While this lower bound appears to generally give small lower bound values, it can also be combined with contraction. Some first tests indicate that this gives in some cases a small improvement to the lower bound obtained with contraction degeneracy.

An interesting and useful method for getting treewidth lower bounds was given by Clautiaux et al. [4]. The procedure from [4] uses another treewidth

lower bound as subroutine. The description in [4] uses the degeneracy, but one can also use another lower bound instead. Our experiments (not given in this extended abstract for space reasons) showed that the contraction degeneracy heuristics generally outperform the method of [4] with degeneracy. Very recently, we combined the method of [4] with the contraction degeneracy heuristics, using a ‘graph improvement step as in [4]’ after each contraction of an edge. While this increases the time of the algorithm significantly, the algorithm still takes polynomial time, and in many cases considerable improvements to the lower bound are obtained. More details of this new heuristic will be given in the full version of this paper.

## 2 Preliminaries

Throughout the paper  $G = (V, E)$  denotes a simple undirected graph. Most of our terminology is standard graph theory/algorithm terminology. As usual, the degree in  $G$  of vertex  $v$  is  $d_G(v)$  or simply  $d(v)$ .  $N(S)$  for  $S \subseteq V$  denotes the open neighbourhood of  $S$ , i.e.  $N(S) = \bigcup_{s \in S} N(s) \setminus S$ . We define:  $\delta(G) := \min_{v \in V} d(v)$ .

**Subgraphs and Minors.** After deleting vertices of a graph and their incident edges, we get an *induced subgraph*. A *subgraph* is obtained, if we additionally allow deletion of edges. If we furthermore allow edge-contractions, we get a *minor*. The treewidth of a minor of  $G$  is at most the treewidth of  $G$  (see e.g. [2]).

**Edge-Contraction.** Contracting edge  $e = \{u, v\}$  in the graph  $G = (V, E)$ , denoted as  $G/e$ , is the operation that introduces a new vertex  $a_e$  and new edges such that  $a_e$  is adjacent to all the neighbours of  $v$  and  $u$  and delete vertices  $u$  and  $v$  and all edges incident to  $u$  or  $v$ :  $G/e := (V', E')$ , where  $V' = \{a_e\} \cup V \setminus \{u, v\}$  and  $E' = \{\{a_e, x\} \mid x \in N(\{u, v\})\} \cup E \setminus \{e' \in E \mid e' \cap e \neq \emptyset\}$ . A *contraction-set* is a cycle free set  $E' \subseteq E(G)$  of edges. Note that after each single edge-contraction the names of the vertices are updated in the graph. Hence, for two adjacent edges  $e = \{u, v\}$  and  $f = \{v, w\}$ , edge  $f$  will be different after contracting edge  $e$ , namely in  $G/e$  we have  $f = \{a_e, w\}$ . Thus,  $f$  represents the same edge in  $G$  and in  $G/e$ . For a contraction-set  $E' = \{e_1, e_2, \dots, e_p\}$ , we define  $G/E' := G/e_1/e_2/\dots/e_p$ . Furthermore, note that the order of edge-contractions to obtain  $G/E'$  is not relevant. A *contraction*  $H$  of  $G$  is a graph such that there exists a contraction-set  $E'$  with:  $H = G/E'$ .

**Treewidth.** A *tree decomposition* of  $G = (V, E)$  is a pair  $(\{X_i \mid i \in I\}, T = (I, F))$ , with  $\{X_i \mid i \in I\}$  a family of subsets of  $V$  and  $T$  a tree, such that  $\bigcup_{i \in I} X_i = V$ , for all  $\{v, w\} \in E$ , there is an  $i \in I$  with  $v, w \in X_i$ , and for all  $i_0, i_1, i_2 \in I$ : if  $i_1$  is on the path from  $i_0$  to  $i_2$  in  $T$ , then  $X_{i_0} \cap X_{i_2} \subseteq X_{i_1}$ . The *width* of tree decomposition  $(\{X_i \mid i \in I\}, T = (I, F))$  is  $\max_{i \in I} |X_i| - 1$ . The *treewidth*  $tw(G)$  of  $G$  is the minimum width among all tree decompositions of  $G$ .



**Degeneracy/MMD.** We also use the term MMD (Maximum Minimum Degree) for the degeneracy. The degeneracy  $\delta D$  of a graph  $G$  is defined to be:  $\delta D(G) := \max_{G'} \{\delta(G') \mid G' \text{ is a subgraph of } G\}$ . The minimum degree of a graph is a lower bound for its treewidth and the treewidth of  $G$  cannot increase by taking subgraphs. The treewidth of  $G$  is at least its degeneracy. (See also [9].)

**Maximum Cardinality Search.** MCS is a method to number the vertices of a graph. It was first introduced by Tarjan and Yannakakis for the recognition of chordal graphs [18]. We start by giving some vertex number 1. In step  $i = 2, \dots, n$ , we choose an unnumbered vertex  $v$  that has the largest number of already numbered neighbours, breaking ties as we wish. Then we associate number  $i$  to vertex  $v$ .

An *MSC-ordering*  $\psi$  can be defined by mapping each vertex to its number:  $\psi(v) := \text{number of } v$ . For a fixed MCS-ordering, let  $v_i := \psi^{-1}(i)$ . The visited degree  $vd_\psi(v_i)$  of  $v_i$  is defined as follows:  $vd_\psi(v) := d_{G[v_1, \dots, v_i]}(v_i)$ . The visited degree  $MCSLB_\psi$  of an MCS-ordering  $\psi$  is defined as follows:  $MCSLB_\psi := \max_{i \in V(G)} vd_\psi(v_i)$

In [12], Lucena has shown that for every graph  $G$  and MCS-ordering  $\psi$  of  $G$ ,  $MCSLB_\psi \leq tw(G)$ . Thus, an MCS numbering gives a lower bound for the treewidth of a graph.

### 3 Contraction Degeneracy

We define the new parameter of contraction degeneracy and the related computational problem. Then we will show the *NP*-completeness of the problem just defined, and consider the complexity of the fixed parameter cases.

#### 3.1 Definition of the Problem

**Definition 1.** The contraction degeneracy  $\delta C$  of a graph  $G$  is defined as follows:

$$\delta C(G) := \max_{G'} \{\delta(G') \mid G' \text{ is a minor of } G\}$$

It is also possible to define  $\delta C$  as the maximum over all contractions of the minimum degree of the contraction. Both definitions are equivalent, since deleting edges or vertices does not help to obtain larger degrees. The corresponding decision problem is formulated as usual:

**Problem:** CONTRACTION DEGENERACY

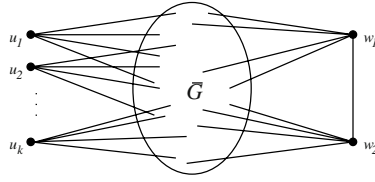
**Instance:** Graph  $G = (V, E)$  and integer  $k \geq 0$ .

**Question:** Is the contraction degeneracy of  $G$  at least  $k$ , i.e. is  $\delta C(G) \geq k$ , i.e. is there a contraction-set  $E' \subseteq E$ , such that  $\delta(G/E') \geq k$ ?

**Lemma 1.** For a graph  $G$ , it holds:  $\delta C(G) \leq tw(G)$

*Proof.* Note that for any minor  $G'$  of  $G$ , we have:  $tw(G') \leq tw(G)$ . Furthermore, for any graph  $G'$ :  $\delta(G') \leq tw(G')$ . The lemma follows now directly.  $\square$





**Fig. 1.** Intermediate graph  $G'$  constructed for the transformation.

### 3.2 NP-Completeness

**Theorem 1.** *The CONTRACTION DEGENERACY problem is NP-complete for bipartite graphs.*

*Proof.* Because of space constraints, we will not give the full proof here. Instead, we only show the construction for the transformation. Clearly, the problem is in NP. The hardness proof is a transformation from the VERTEX COVER problem. Let be given a VERTEX COVER instance  $(G, k)$ , with  $G = (V, E)$ .

*Construction:* Starting from the complement of  $G$ , we add vertices and edges to construct a new graph  $G'$ , which is an intermediate step in the transformation.  $G' = (V', E')$  is formally defined as follows, see Figure 1:

$$\begin{aligned} V' &= V \cup \{w_1, w_2\} \cup \{u_1, \dots, u_k\} \\ E' &= (V \times V \setminus E) \cup \{ \{w_1, w_2\} \} \cup \{ \{w_i, v\} \mid i \in \{1, 2\} \wedge v \in V \} \cup \\ &\quad \{ \{u_i, v\} \mid i \in \{1, \dots, k\} \wedge v \in V \} \end{aligned}$$

The final graph  $G^* = (V^*, E^*)$  in our construction is obtained by subdividing any edge in  $G'$ , i.e. replacing each edge in  $G'$  by a path with two edges.

$$V^* = V' \cup \{v_e \mid e \in E'\} \text{ and } E^* = \{ \{u, v_e\}, \{v_e, w\} \mid e = \{u, w\} \in E' \}$$

Let  $n := |V|$ , the constructed instance of the CONTRACTION DEGENERACY problem is  $(G^*, n + 1)$ .

Now, it is easy to see that  $G^*$  is a bipartite graph. It is also possible to prove that there is a vertex cover for  $G$  of size at most  $k$  if, and only if  $\delta C(G^*) \geq n + 1$ .  $\square$

### 3.3 Fixed Parameter Cases

Now, we consider the fixed parameter case of the CONTRACTION DEGENERACY problem. I.e., for a fixed integer  $k$ , we consider the problem to decide for a given graph  $G$  if  $G$  has a contraction with minimum degree  $k$ . Graph minor theory gives a fast answer to this problem. Those unfamiliar with this theory are referred to [6].

**Theorem 2.** *The CONTRACTION DEGENERACY problem can be solved in linear time when  $k$  is a fixed integer with  $k \leq 5$ , and can be solved in  $O(n^3)$  time when  $k$  is a fixed integer with  $k \geq 6$ .*

*Proof.* Let  $k$  be a fixed integer. Consider the class of graphs  $\mathcal{G}_k = \{G \mid G \text{ has contraction degeneracy at most } k - 1\}$ .  $\mathcal{G}_k$  is closed under taking minors: if  $H$  is a minor of  $G$  and  $H$  has contraction degeneracy at least  $k$ , then  $G$  has also contraction degeneracy at least  $k$ . As every class of graphs that is closed under minors has an  $O(n^3)$  algorithm to test membership by Robertson-Seymour graph minor theory (see [6]), the theorem for the case that  $k \geq 6$  follows.

Suppose now that  $k \leq 5$ . There exists a planar graph  $G_k$  with minimum degree  $k$ . Hence,  $G_k \notin \mathcal{G}_k$ . A class of graphs that is closed under taking minors and does not contain all planar graphs has a linear time membership test (see [6]), which shows the result for the case that  $k \leq 5$ .  $\square$

It can be noted that the cases that  $k = 1$  and  $k = 2$  are very simple: a graph has contraction degeneracy at least 1, if and only if it has at least one edge, and it has contraction degeneracy at least 2, if and only if it is not a forest. The result is non-constructive when  $k \geq 6$ ; when  $k \leq 5$ , the result can be made constructive by observing that the property that  $G$  has contraction degeneracy  $k$  can be formulated in monadic second order logic for fixed  $k$ . Thus, we can solve the problem as follows: the result of [17] applied to  $G_k$  gives an explicit upper bound  $c_k$  on the treewidth of graphs in  $\mathcal{G}_k$ . Test if  $G$  has treewidth at most  $c_k$ , and if so, find a tree decomposition with width at most  $c_k$  with the algorithm of [1]. If  $G$  has treewidth at most  $c_k$ , use the tree decomposition to test if the MSOL formula holds for  $G$  [5]; if not, we directly know that  $G$  has contraction degeneracy at least  $k$ . The constant factors hidden in the  $O$ -notation are very large, thus these results have only theoretical importance. We summarise the different cases in the following table.

**Table 1.** Complexity of CONTRACTION DEGENERACY

$k$	Time	Reference
1	$O(n)$	Trivial
2	$O(n)$	$G$ is not a forest
3, 4, 5	$O(n)$	[1,5,17],MSOL
fixed $k \geq 6$	$O(n^3)$	[16,15]
variable $k$	$NP$ -complete	Theorem 1

### 4 Maximum Cardinality Search with Contraction

From a maximum cardinality search ordering, we can derive a parameter that is a lower bound for the treewidth of a graph (see Section 2). We define four problems related to this parameter. Each of these is  $NP$ -complete or  $NP$ -hard, respectively. We consider the following problem, and variants.

**Problem:** MCSLB WITH CONTRACTION

**Instance:** Graph  $G = (V, E)$ , integer  $k$ .

**Question:** Does  $G$  have a contraction  $H$ , and  $H$  an MCS-ordering  $\psi$  with the visited degree of  $\psi$  at least  $k$ ?

In the variant MCSLB WITH MINORS we require instead that  $H$  is a minor of  $G$ . In the variants MINMCSLB WITH CONTRACTION and MINMCSLB WITH MINORS we ask that *every* MCS-ordering  $\psi$  of  $H$  has visited degree at least  $k$ .

**Theorem 3.** MCSLB WITH CONTRACTION and MCSLB WITH MINORS are NP-complete. MINMCSLB WITH CONTRACTION and MINMCSLB WITH MINORS are NP-hard.

The hardness proofs are transformations from VERTEX COVER, with a somewhat more involved variant of the proof of Theorem 1. Membership in NP is trivial for MCSLB WITH CONTRACTION and MCSLB WITH MINORS, but unknown for the other two problems.

The fixed parameter case of MCSLB WITH MINORS can be solved in linear time with help of graph minor theory. Observing that the set of graphs  $\{G \mid G \text{ does not have a minor } H, \text{ such that } H \text{ has an MCS-ordering } \psi \text{ with the visited degree of } \psi \text{ at least } k\}$  is closed under taking of minors, and does not include all planar graphs (see [3]), gives us by the Graph Minor theorem of Robertson and Seymour and the results in [1,17] the following result.

**Theorem 4.** MCSLB WITH MINORS and MINMCSLB WITH MINORS are linear time solvable for fixed  $k$ .

The fixed parameter cases of MINMCSLB WITH MINORS give an  $O(n^3)$  algorithm (linear when  $k \leq 5$ ), similar as for Theorem 2. Again, note that these results are only of theoretical interest, due to the high constant factors hidden in the  $O$ -notation, and the non-constructiveness of the proof.

## 5 Experimental Results

In this section, we report on the results of computational experiments we have carried out. We tested our algorithms on a number of graphs, obtained from two application areas where treewidth plays a role in algorithms that solve the problems at hand. The first set of instances are probabilistic networks from existing decision support systems from fields like medicine and agriculture. The second set of instances are from frequency assignment problems from the EUCLID CALMA project (see e.g. [8]). We have also used this set of instances in earlier experiments. We excluded those networks for which the MMD heuristic already gives the exact treewidth. Some of the graphs can be obtained from [19]. All algorithms have been written in C++, and the computations have been carried out on a Linux operated PC with a 3.0 GHz Intel Pentium 4 processor.

Table 2. Graph sizes, upper bounds, and MMD/MMD+ lower bounds

instance	size		UB	MMD		MMD+		
	V	E		LB	CPU	min-d. LB CPU	max-d. LB CPU	least-c. LB CPU
barley	48	126	7	5	0.00	6 0.00	5 0.00	6 0.00
<b>diabetes</b>	413	819	4	3	0.00	4 0.01	4 0.00	4 0.00
link	724	1738	13	4	0.00	8 0.02	5 0.01	11 0.03
<b>mildew</b>	35	80	4	3	0.00	4 0.00	3 0.00	4 0.00
munin1	189	366	11	4	0.00	8 0.01	5 0.00	10 0.00
munin2	1003	1662	7	3	0.01	6 0.01	4 0.01	6 0.02
<b>munin3</b>	1044	1745	7	3	0.01	7 0.01	4 0.02	7 0.02
munin4	1041	1843	8	4	0.01	7 0.01	5 0.01	7 0.02
oesoca+	67	208	11	9	0.00	9 0.00	9 0.00	9 0.00
oow-trad	33	72	6	3	0.00	4 0.00	4 0.00	5 0.00
<b>oow-bas</b>	27	54	4	3	0.00	4 0.00	3 0.00	4 0.00
oow-solo	40	87	6	3	0.00	4 0.00	4 0.00	5 0.00
<b>pathfinder</b>	109	211	6	5	0.00	6 0.00	5 0.00	6 0.01
pignet2	3032	7264	135	4	0.01	29 0.11	10 0.07	38 0.20
pigs	441	806	10	3	0.00	6 0.01	4 0.00	7 0.01
ship-ship	50	114	8	4	0.00	6 0.00	4 0.00	6 0.00
water	32	123	10	6	0.00	7 0.00	7 0.00	8 0.00
<b>wilson</b>	21	27	3	2	0.00	3 0.00	3 0.00	3 0.00
celar01	458	1449	17	8	0.00	12 0.01	9 0.00	14 0.02
<b>celar02</b>	100	311	10	9	0.00	9 0.00	9 0.00	10 0.00
celar03	200	721	15	8	0.00	11 0.00	9 0.00	13 0.01
celar04	340	1009	16	9	0.00	12 0.01	9 0.00	13 0.01
celar05	200	681	15	9	0.01	11 0.00	9 0.00	13 0.01
<b>celar06</b>	100	350	11	10	0.00	11 0.00	10 0.00	11 0.01
celar07	200	817	18	11	0.00	13 0.00	12 0.01	15 0.00
celar08	458	1655	18	11	0.00	13 0.01	12 0.01	15 0.01
celar09	340	1130	18	11	0.01	13 0.01	12 0.00	15 0.01
celar11	340	975	15	8	0.00	11 0.00	9 0.00	13 0.01
graph01	100	358	25	8	0.00	9 0.01	9 0.00	14 0.00
graph02	200	709	51	6	0.00	11 0.00	7 0.01	20 0.02
graph03	100	340	22	5	0.00	8 0.00	6 0.01	14 0.00
graph04	200	734	55	6	0.00	12 0.01	7 0.00	19 0.02
graph05	100	416	26	8	0.00	9 0.00	9 0.00	15 0.00
graph06	200	843	53	8	0.00	12 0.01	9 0.00	21 0.02
graph07	200	843	53	8	0.00	12 0.01	9 0.00	21 0.02
graph08	340	1234	91	7	0.00	16 0.02	8 0.01	26 0.04
graph09	458	1667	118	8	0.01	17 0.03	9 0.01	29 0.06
graph10	340	1275	96	6	0.00	15 0.01	7 0.01	27 0.04
graph11	340	1425	98	7	0.00	17 0.01	8 0.01	27 0.05
graph12	340	1256	90	5	0.00	16 0.01	6 0.01	25 0.04
graph13	458	1877	126	6	0.00	18 0.02	7 0.01	31 0.07
graph14	458	1398	121	4	0.00	20 0.02	8 0.02	27 0.05

**Table 3.** MCSLB/MCSLB+ lower bounds

instance	MCSLB		MCSLB+ LBs						
			min-deg.		last-mcs		max-mcs		average
	LB	CPU	min-d.	least-c.	min-d.	least-c.	min-d.	least-c.	
barley	5	0.01	6	6	6	6	6	5	0.06
diabetes	4	0.92	4	4	4	4	4	4	5.23
link	5	3.09	8	10	8	11	8	6	43.08
mildew	3	0.00	4	4	4	4	4	4	0.03
munin1	4	0.17	8	10	9	10	9	7	0.95
munin2	4	5.46	6	6	6	6	5	6	31.30
munin3	4	5.87	6	7	7	7	6	7	33.80
<b>munin4</b>	5	6.06	7	7	7	8	6	7	48.30
oesoca+	9	0.02	9	9	9	9	9	9	0.15
oow-trad	4	0.00	5	5	5	5	5	4	0.03
oow-bas	3	0.00	4	4	4	4	4	4	0.02
oow-solo	4	0.01	4	5	4	5	5	5	0.05
pathfinder	6	0.05	6	6	6	6	6	6	0.34
pignet2	5	59.60	28	39	30	39	16	18	509.60
pigs	3	1.01	7	7	7	7	6	6	5.12
ship-ship	5	0.01	6	6	6	6	6	6	0.06
water	8	0.00	8	8	8	8	8	8	0.04
wilson	3	0.00	3	3	3	3	3	3	0.01
celar01	10	1.20	12	13	12	14	12	13	17.08
celar02	9	0.06	9	10	9	10	9	9	0.30
celar03	9	0.23	11	12	11	13	12	12	1.54
celar04	11	0.66	11	13	12	13	12	13	4.85
celar05	9	0.23	12	13	12	13	12	12	1.80
celar06	11	0.06	11	11	11	11	11	11	0.33
celar07	12	0.24	13	15	13	15	13	15	1.66
celar08	12	1.45	13	15	14	15	13	15	17.04
celar09	12	0.82	13	15	14	15	14	15	4.62
celar11	10	0.66	11	13	12	13	12	12	4.43
graph01	9	0.06	9	15	11	14	12	13	0.45
graph02	8	0.24	12	19	12	19	14	13	2.54
graph03	6	0.06	9	13	10	14	9	13	0.49
graph04	8	0.25	12	20	13	20	14	16	1.95
graph05	9	0.06	10	15	11	16	11	14	0.47
graph06	9	0.26	12	22	14	22	14	15	2.22
graph07	9	0.26	12	22	14	22	14	15	2.15
graph08	9	0.76	17	26	18	26	18	20	5.89
graph09	9	1.43	17	30	20	28	22	23	11.51
graph10	8	0.77	18	26	17	26	19	22	6.25
graph11	8	0.80	15	27	18	27	17	26	6.53
graph12	7	0.76	15	25	16	25	17	21	5.92
graph13	8	1.48	18	32	19	31	20	28	12.89
graph14	5	1.35	19	28	20	28	21	25	10.11

## 5.1 The Heuristics

**MMD** computes exactly the degeneracy, by deleting a vertex of minimum degree and its incident edges in each iteration.

**MMD+** is a derivative of MMD. Instead of deleting a vertex  $v$  of minimum degree, we contract it to a neighbour  $u$ . We consider three strategies how to select a neighbour:

- *min-d.* selects a neighbour with minimum degree, motivated by the idea that the smallest degree is increased as fast as possible in this way.
- *max-d.* selects a neighbour with maximum degree, motivated by the idea that we end up with some vertices of very high degree.
- *least-c.* selects a neighbour  $u$  of  $v$ , such that  $u$  and  $v$  have the least number of common neighbours, motivated by the idea to delete as few as possible edges in each iteration to get a high minimum degree.

**MCSLB** computes  $|V|$  *MCS*-orderings  $\psi$  – one for each vertex as start vertex. It returns the maximum over these orderings of  $MCSLB_\psi$ , cf. [3].

**MCSLB+** initially uses MCSLB to find a start vertex  $w$  with largest  $MCSLB_\psi$ . After each computation of an *MCS*-ordering, we select a vertex which we will contract, and then we apply MCSLB+ again. To reduce the CPU time consumption,  $MCSLB_\psi$  is computed only for start vertex  $w$  instead of all possible start vertices. Three strategies for selecting a vertex  $v$  to be contracted are examined:

- *min-deg.* selects a vertex of minimum degree.
- *last-mcs* selects the last vertex in the just computed *MCS*-ordering.
- *max-mcs* selects a vertex with maximum visited degree in the just computed *MCS*-ordering.

Once a vertex  $v$  is selected, we select a neighbour  $u$  of  $v$  using the two strategies *min-d.* and *least-c.* that are already explained for MMD+.

## 5.2 Tables

Table 2 and 3 show the computational results for the described graphs. Table 2 shows the size of the graphs and the best known upper bound UB for treewidth, cf. [9,19]. Furthermore, Table 2 presents the lower bounds LB and the CPU times in seconds for the MMD and MMD+ heuristic with different selection strategies. Table 3, shows the same information for the MCSLB and MCSLB+ heuristics with different combinations of selection strategies. Because of space constraints and similarity, we only give the average running times.

In both tables, the graphs for which the lower and upper bound coincide are highlighted bold. In total, the treewidth of 8 graphs could be determined by MMD+, and one more by MCSLB+.

The MMD+ results show that the *min-d.* and in particular the *least-c.* strategy are very successful. The *max-d.* strategy is only marginal better than the degeneracy. For the MCSLB+ we observe that *min-deg.* and *last-mcs* in combination with *least-c.* outperform the other strategies. The added value of MCSLB+ to MMD+ is marginal, also because of the relatively large computation times.

The success of the *least-c.* strategy is explained as follows. When we contract an edge  $\{v, w\}$ , for each common neighbor  $x$  of  $v$  and  $w$ , the two edges  $\{v, x\}$  and  $\{w, x\}$  become the same edge in the new graph. Thus, with the *least-c.* strategy, we have more edges in the graph after contraction, which often leads to a larger minimum degree.

## 6 Discussion and Concluding Remarks

In this article, we examined two new methods MMD+ and MCSLB+ for treewidth lower bounds which are derivatives of the MMD and MCSLB heuristics. We showed the two corresponding decision problems to be *NP*-complete.

The practical experiments show that contracting edges is a very good approach for obtaining lower bounds for treewidth as it considerably improves known lower bounds. When contracting a vertex, it appears that the best is to contract to a neighbor that has as few as possible common neighbors. We can see that in some cases the MCSLB+ gives a slightly better lower bound than the MMD+. However, the running times of the MMD+ heuristic are almost always negligible, while the running times of the MCSLB+ heuristic can reach a considerable order of magnitude. Furthermore, we see that the strategy for selecting a neighbour  $u$  of  $v$  with the least number of common neighbours of  $u$  and  $v$  often performs best.

Finally, notice that although the gap between lower and upper bound could be significantly closed by contracting edges within the algorithms, the absolute gap is still large for many graphs (pignet2, graph\*), which asks for research on new and better heuristics.

MMD+ is a simple method usually performing very well. However, it has its limits, e.g., on planar graphs. A planar graph always has a vertex of degree  $\leq 5$ , and therefore MMD and MMD+ can never give lower bounds larger than 5. Similar behaviour could be observed in experiments on ‘almost’ planar graphs.

Apart from its function as a treewidth lower bound, the contraction degeneracy appears to be an attractive and elementary graph measure, worth further study. For instance, interesting topics are its computational complexity on special graph classes or the complexity of approximation algorithms with a guaranteed performance ratio. While we know the fixed parameter cases are polynomial time solvable, it is desirable to have algorithms that solve e.g., contraction degeneracy for small values of  $k$  efficiently in practice.

## References

1. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25:1305–1317, 1996.
2. H. L. Bodlaender. A partial  $k$ -arboretum of graphs with bounded treewidth. *Theor. Comp. Sc.*, 209:1–45, 1998.
3. H. L. Bodlaender and A. M. C. A. Koster. On the maximum cardinality search lower bound for treewidth, 2004. Extended abstract to appear in proceedings WG 2004.
4. F. Clautiaux, J. Carlier, A. Moukrim, and S. Négre. New lower and upper bounds for graph treewidth. In J. D. P. Rolim, editor, *Proceedings International Workshop on Experimental and Efficient Algorithms, WEA 2003*, pages 70–80. Springer Verlag, Lecture Notes in Computer Science, vol. 2647, 2003.
5. B. Courcelle. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.
6. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1998.
7. V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. To appear in proceedings UAI’04, Uncertainty in Artificial Intelligence, 2004.
8. A. M. C. A. Koster. *Frequency assignment - Models and Algorithms*. PhD thesis, Univ. Maastricht, Maastricht, the Netherlands, 1999.
9. A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel. Treewidth: Computational experiments. In H. Broersma, U. Faigle, J. Hurink, and S. Pickl, editors, *Electronic Notes in Discrete Mathematics*, volume 8. Elsevier Science Publishers, 2001.
10. A. M. C. A. Koster, S. P. M. van Hoesel, and A. W. J. Kolen. Solving partial constraint satisfaction problems with tree decomposition. *Networks*, 40:170–180, 2002.
11. S. J. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *The Journal of the Royal Statistical Society. Series B (Methodological)*, 50:157–224, 1988.
12. B. Lucena. A new lower bound for tree-width using maximum cardinality search. *SIAM J. Disc. Math.*, 16:345–353, 2003.
13. S. Ramachandramurthi. A lower bound for treewidth and its consequences. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Proceedings 20th International Workshop on Graph Theoretic Concepts in Computer Science WG’94*, pages 14–25. Springer Verlag, Lecture Notes in Computer Science, vol. 903, 1995.
14. S. Ramachandramurthi. The structure and number of obstructions to treewidth. *SIAM J. Disc. Math.*, 10:146–157, 1997.
15. N. Robertson and P. D. Seymour. Graph minors. XX. Wagner’s conjecture. To appear.
16. N. Robertson and P. D. Seymour. Graph minors. XIII. The disjoint paths problem. *J. Comb. Theory Series B*, 63:65–110, 1995.
17. N. Robertson, P. D. Seymour, and R. Thomas. Quickly excluding a planar graph. *J. Comb. Theory Series B*, 62:323–348, 1994.
18. R. E. Tarjan and M. Yannakakis. Simple linear time algorithms to test chordality of graphs, test acyclicity of graphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13:566–579, 1984.
19. Treewidthlib. <http://www.cs.uu.nl/people/hansb/treewidthlib>, 2004-03-31.



# Load Balancing of Indivisible Unit Size Tokens in Dynamic and Heterogeneous Networks<sup>\*</sup>

Robert Elsässer, Burkhard Monien, and Stefan Schamberger

Institute for Computer Science  
University of Paderborn  
Fürstenallee 11, D-33102 Paderborn  
{elsa,bm,schaum}@uni-paderborn.de

**Abstract.** The task of balancing dynamically generated work load occurs in a wide range of parallel and distributed applications. Diffusion based schemes, which belong to the class of nearest neighbor load balancing algorithms, are a popular way to address this problem. Originally created to equalize the amount of arbitrarily divisible load among the nodes of a static and homogeneous network, they have been generalized to heterogeneous topologies. Additionally, some simple diffusion algorithms have been adapted to work in dynamic networks as well. However, if the load is not divisible arbitrarily but consists of indivisible unit size tokens, diffusion schemes are not able to balance the load properly. In this paper we consider the problem of balancing indivisible unit size tokens on dynamic and heterogeneous systems. By modifying a randomized strategy invented for homogeneous systems, we can achieve an asymptotically minimal expected overload in  $l_1$ ,  $l_2$  and  $l_\infty$  norm while only slightly increasing the run-time by a logarithmic factor. Our experiments show that this additional factor is usually not required in applications.

## 1 Introduction

Load balancing is a very important prerequisite for the efficient use of parallel computers. Many distributed applications produce work load dynamically which often leads to dramatical differences in runtime. Thus, in order to achieve an efficient use of the processor network, the amount of work has to be balanced during the computation. Obviously, we can ensure an overall benefit only if the balancing scheme itself is highly efficient.

If load is arbitrarily divisible, the balancing problem for a homogeneous network with  $n$  nodes can be described as follows. At the beginning, each node  $i$  contains some work load  $w_i$ . The goal is to obtain the balanced work load  $\bar{w} = \sum_{i=1}^n w_i/n$  on all nodes. We assume for now that no load is generated or consumed during the balancing process, i.e., we consider a static load balancing scenario.

---

<sup>\*</sup> This work was partly supported by the German Science Foundation (DFG) project SFB-376 and by the IST Program of the EU under contract numbers IST-1999-14186 (ALCOM-FT) and IST-2001-33116 (FLAGS).

A popular class of load balancing algorithms consists of diffusion schemes e.g. [1]. They work iteratively and each node migrates a load fraction (flow) over the topology's communication links (edges), depending on the work load difference to its neighbors. Hence, these schemes operate locally and therefore avoid expensive global communication. Formally, if we denote the load of node  $i$  after  $k$  iterations by  $w_i^k$  and the flow over edge  $\{i, j\}$  in step  $k$  by  $y_{i,j}^{k-1}$ , then

$$\begin{aligned} \forall e = \{i, j\} \in E : y_{i,j}^{k-1} &= \alpha_{i,j}(w_i^{k-1} - w_j^{k-1}); \\ \text{and } w_i^k &= w_i^{k-1} - \sum_{\{i,j\} \in E} y_{i,j}^{k-1}, \end{aligned} \quad (1)$$

is computed where all  $\alpha_{i,j}$  satisfy the conditions described in the next Section. Equation 1 can be rewritten in matrix notation as  $w^k = Mw^{k-1}$ , where the matrix  $M$  is called the *Diffusion Matrix* of the network. This algorithm is called *First Order Scheme* (FOS) and converges to a balanced state computing the  $l_2$ -minimal flow. Improvements of FOS are the *Second Order Schemes* (SOS) which perform faster than FOS by almost a quadratic factor [2,3].

In [4], these schemes are extended to heterogeneous networks consisting of processors with different computing powers. In such an environment, computations perform faster if the load is balanced proportionally to the nodes' computing speed  $s_i$ :

$$\overline{w_i} := \frac{\sum_{i=1}^n w_i}{\sum_{i=1}^n s_i} s_i. \quad (2)$$

Obviously, this is a generalization of the load balancing problem in homogeneous networks ( $s_i = 1$ ). Diffusion in networks with communication links of different capacities are analyzed e.g. in [3,5]. It is shown that the existing balancing schemes can be generalized, such that roughly speaking faster communication links get a higher load migration volume than slower ones. The two generalizations can be combined as described in [4]. Heterogeneous topologies are extremely attractive because they often appear in real hardware installations containing machines and networks of different capabilities. In [6] it is demonstrated that FOS is also applicable to balance load in dynamic networks where edges fail from time to time or are present depending on the distance of the moving nodes.

In contrast to the above implementations, work load in real world applications usually cannot be divided arbitrarily often, but only to some extend. Thus, to address the load balancing problem properly, a more restrictive model is needed. The unit-size token model [2] assumes a smallest load entity, the *unit-size token*. Furthermore, work load is always represented by a multiple of this smallest entity. It is clear that load in this model is usually not completely balanceable. Unfortunately, as shown for FOS in homogeneous networks in e.g. [2, 7,8], the use of integer values prevents the known diffusion schemes to balance the load completely. Especially if only a relatively small amount of tokens exists, a considerable load difference remains.

To measure the balance quality in a network, usually two metrics are considered. The  $l_2$ -norm expresses how well the load is distributed on the pro-

cessors, hence the error in this norm  $\|e\|_2 = (\sum_{i=1}^n (w_i^k - \bar{w}_i)^2)^{1/2}$  should be minimized. One is even more interested in minimizing the overall computation time and therefore the load in  $l_\infty$ -norm, which is equivalent to minimizing the maximal (weighted) load occurring on the processors. Recall, that if the error  $\|e\|_\infty = \max |\bar{w} - w_i|$  is less than  $b$ , then  $\|e\|_2$  is bounded by  $\sqrt{n} \cdot b$ .

The problem of balancing indivisible tokens in homogeneous networks is addressed e.g. in [9]. The proposed algorithm improves the local balance significantly, but still cannot guarantee a satisfactory global balance. The authors of [2] use a discretization of FOS by transmitting the flow  $y_{i,j} = \lfloor \alpha_{i,j} (w_i^{k-1} - w_j^{k-1}) \rfloor$  over edge  $\{i, j\}$  in step  $k$ . After  $k = \Theta((d/\lambda_2) \log(En))$  steps, the error in  $l_2$ -norm is reduced to  $\mathcal{O}(nd^2/\lambda_2)$ , where  $E = \max_i |w_i^0 - \bar{w}_i|$  is the *maximal initial load imbalance* in the network,  $\lambda_2$  denotes the second smallest eigenvalue of the Laplacian of the graph (see next Section), and  $d$  is the maximal vertex degree. This bound has been improved by showing that the error in  $l_\infty$ -norm is  $\mathcal{O}((d^2/\lambda_2) \log(n))$  after  $k = \Theta((d/\lambda_2) \log(En))$  iterations. [7]. Furthermore, there exist a number of results to improve the balance on special topologies, e.g. [10,11,12]. The randomized algorithm developed in [8] reduces  $\|e^k\|_2$  to  $\mathcal{O}(\sqrt{n})$  in  $k = \Theta((d/\lambda_2) (\log(E) + \log(n) \log(\log(n))))$  iteration steps with high probability, if  $\bar{w}_i$  exceeds some certain threshold. Since it is also shown that in  $l_2$ -norm  $\mathcal{O}(\sqrt{n})$  is the best expected asymptotic result that can be achieved concerning the final load distribution [8], this algorithm provides an asymptotic optimal result. However, concerning the minimization of the overall computation time, the above algorithm only guarantees  $\|w^k\|_\infty \leq \bar{w}_i + \mathcal{O}(\log(n))$ , even if  $k \rightarrow \infty$ . Therefore, this algorithm does not achieve an optimal situation concerning the maximal overload in the system.

In this paper, we present a modification of the randomized algorithm from [8] in order to minimize the overload in the system in  $l_1$ ,  $l_2$ - and  $l_\infty$ -norm, respectively. We show that this algorithm can also be applied in heterogeneous networks, achieving the corresponding asymptotic overload for the weighted  $l_1$ -,  $l_2$ -, or  $l_\infty$ -norms (for a definition refer to Section 3). The algorithm's run-time increases slightly by at most a logarithmic factor and is bounded by  $\Theta((d/\lambda_2) \cdot (\log(E) + \log^2(n)))$  with high probability, whenever  $\bar{w}_i$  exceeds some threshold. Note, that from our experiments we can conclude that the additional run-time is usually not needed. Furthermore, our results can be generalized to dynamic networks which are modeled as graphs with a constant node set, but a varying set of edges during the iterations. Similar dynamic networks have been used to analyze the token distribution problem in [13]. However, in contrast to diffusion, the latter only allows one single token to be sent over an edge in each iteration.

The outline of the paper is as follows. In the next Section, we give an overview of the basic definitions and the necessary theoretical background. Then, we describe the randomized strategy for heterogeneous networks and show that this algorithm minimizes the overload to the asymptotically best expected value whenever the heterogeneity obeys some restrictions. In Section 4, we present some experimental results and finally Section 5 contains our conclusion.

## 2 Background and Definitions

Let  $G = (V, E)$  be an undirected, connected, weighted graph with  $|V| = n$  nodes and  $|E| = m$  edges. Let  $c_{i,j} \in \mathbb{R}^N$  be the *capacity* of edge  $e_{i,j} \in E$ ,  $s_i$  be the *processing speed* of node  $i \in V$ , and  $w_i \in \mathbb{R}$  be its *work load*. In case of indivisible load entities, this value represents the number of unit-size tokens.

Let  $A \in \mathbb{R}^{n \times n}$  be the *Weighted Adjacency Matrix* of  $G$ . As  $G$  is undirected,  $A$  is symmetric. Column/row  $i$  of  $A$  contains  $c_{i,j}$  where  $j$  and  $i$  are neighbors in  $G$ . For some of our constructions we need the *Laplacian*  $L \in \mathbb{Z}^{n \times n}$  of  $G$  defined as  $L := D - A$ , where  $D \in \mathbb{N}^{n \times n}$  contains the weighted degrees as diagonal entries, e.g.  $D_{i,i} = \sum_{\{i,j\} \in E} c_{i,j}$ , and 0 elsewhere.

The Laplacian  $L$  and its eigenvalues are used to analyze the behavior of diffusion in homogeneous networks. For heterogeneous networks we have to apply the generalized Laplacian  $LS^{-1}$ , where  $S \in \mathbb{R}^{n \times n}$  denotes the diagonal matrix containing the processor speeds  $s_i$ . We assume that  $1 \leq s_i \leq \mathcal{O}(n^\delta)$  with  $\delta < 1$ .

Generalizing the local iterative algorithm of Equation 1 in the case of arbitrary divisible tokens to heterogeneous networks, one yields [4]:

$$w_i^k = w_i^{k-1} - \sum_{j \in N(i)} c_{i,j} \left( \frac{w_i^{k-1}}{s_i} - \frac{w_j^{k-1}}{s_j} \right) \quad (3)$$

Here,  $N(i)$  denotes the set of neighbors of  $i \in V$ , and  $w_i^k$  is the load of the node  $i$  after the  $k$ -th iteration. As mentioned, this scheme is known as FOS and converges to the average load  $\bar{w}$  of Equation 2 [4]. It can be written in matrix notation as  $w^k = Mw^{k-1}$  with  $M = I - LS^{-1} \in \mathbb{R}^{n \times n}$ .  $M$  contains  $c_{i,j}/s_j$  at position  $(i, j)$  for every edge  $e = \{i, j\}$ ,  $1 - \sum_{e=\{i,j\} \in E} c_{i,j}/s_j$  at diagonal entry  $i$ , and 0 elsewhere.

Now, let  $\lambda_i$ ,  $1 \leq i \leq n$  be the eigenvalues of the Laplacian  $LS^{-1}$  in increasing order. It is known that  $\lambda_1 = 0$  with eigenvector  $(s_1, \dots, s_n)$  [14]. The values of  $c_{i,j}$  have to be chosen such that the diffusion matrix  $M$  has the eigenvalues  $-1 < \mu_i = 1 - \lambda_i \leq 1$ . Since  $G$  is connected, the first eigenvalue  $\mu_1 = 1$  is simple with eigenvector  $(s_1, \dots, s_n)$ . We denote by  $\gamma = \max\{|\mu_2|, |\mu_n|\} < 1$  the second largest eigenvalue of  $M$  according to absolute values and call it the *diffusion norm* of  $M$ .

Several modifications to FOS have been discussed in the past. One of them is SOS [15], which has the form

$$w^1 = Mw^0, w^k = \beta Mw^{k-1} + (1 - \beta)w^{k-2}, \quad k = 2, 3, \dots \quad (4)$$

Setting  $\beta = 2/(1 + \sqrt{1 - \mu_2^2})$  results in the fastest convergence.

As described in the introduction, we denote the *error* after  $k$  iteration steps by  $e^k$ , where  $e^k = w^k - \bar{w}$ . The convergence rate of diffusion schemes in the case of arbitrary divisible tokens depends on how fast the system becomes  $\epsilon$ -balanced, i.e., the final error  $\|e^k\|_2$  is less than  $\epsilon\|e^0\|_2$ . By using simple calculations and the results from [2,4], we obtain the following lemma.

**Lemma 1.** *Let  $G$  be a graph and  $L$  its Laplacian, where  $c_{i,j} = 1/(c \max\{d_i, d_j\})$ ,  $d_i$  is the degree of node  $i$  and  $c \in (1, 2]$  a constant. Let  $M = I - LS^{-1}$  be the diffusion matrix of  $G$  and set  $\beta = 2/(1 + \sqrt{1 - \gamma^2})$ . Then, FOS and SOS require  $\Theta((d/\lambda_2) \cdot \ln(s_{\max}/(\epsilon s_{\min})))$  and  $\Theta(\sqrt{d/\lambda_2} \cdot \ln(s_{\max}/(\epsilon s_{\min})))$  steps, respectively, to  $\epsilon$ -balance the system, where  $s_{\max} = \max_i s_i$  and  $s_{\min} = \min_i s_i$ .*

Although SOS converges faster than FOS by almost a quadratic factor and therefore seems preferable, it has a drawback. During the iterations, the outgoing flow of a node  $i$  may exceed  $w_i$  which results in negative load. On static networks, the two phase model copes with this problem [3]. The first phase determines the balancing flow while the second phase then migrates the load accordingly. However, this two phase model cannot be applied on dynamic networks because edges included in the flow computation might not be available for migration.

Even in cases where the nodes' outgoing flow does not exceed their load, we have not been able to show the convergence of SOS in dynamic systems yet. Hence, our diffusion scheme analysis on such networks is restricted to FOS.

Since the diffusion matrix  $M$  is double stochastic, FOS can also be interpreted as a Markov process, where  $M$  is the corresponding transition matrix and the balanced load situation is the related stationary distribution. Using this Markov chain approach, the following lemma can be stated [16,17].

**Lemma 2.** *Let  $G$  be a graph and we assume that a token performs a random walk on  $G$  according to the diffusion matrix  $M$ . If the values  $s_i$  are in a range  $[1, n^\delta]$ ,  $\delta \in [0, 1]$ , then a constant  $a$  exists such that after  $a((d \cdot \log(n))/\lambda_2)$  steps the probability  $p_i$  of the token being situated on some node  $i$  is bounded by  $(s_i / \sum_{j=1}^n s_j) - (1/n^4) \leq p_i \leq (s_i / \sum_{j=1}^n s_j) + (1/n^4)$ .*

### 3 The Randomized Algorithm

In this Section, we describe a randomized algorithm that guarantees a final weighted overload of  $\mathcal{O}(1)$  in  $l_\infty$ -norm for the load balancing problem of indivisible unit-size tokens in heterogeneous networks.

First, we show that FOS as well as SOS do not guarantee a satisfactory balance in case of unit size tokens on inhomogeneous networks. As mentioned, the outgoing load from some node  $i$  may exceed  $w_i$  when applying SOS. Hence, simply rounding down the flow to the nearest integer does not lead to the desired algorithm. Therefore, the authors of [2] introduced a so called 'I Owe You' (IOU) unit on each edge. These IOUs represent the difference between the total flow moved along an edge in the idealized scheme and in the realistic algorithm. If the IOUs accumulate, then we can not prove any result w.r.t. the convergence of the adapted SOS. Nevertheless, in most of the simulations on static networks, even if the IOUs emerge, the number of owed units becomes very small after a few rounds.

If we assume that the IOUs tend to zero after a few iterations, then we can use the techniques of [18] to state the following theorem.

**Theorem 1.** Let  $G = (V, E)$  be a node weighted graph, where  $s_i$  is the weight of node  $i \in V$ , and let  $w^0$  be the initial load distribution on  $G$ . Executing SOS on  $G$ , we assume that after a few iteration steps the IOUs are bounded by some constant on each edge of the graph. Then, after sufficient number of steps  $k$ , the error  $\|e^k\|_2$  is  $\mathcal{O}((\sqrt{n \cdot s_{\max}} \cdot d)/(\sqrt{s_{\min}} \cdot (1 - \gamma)))$ , where  $\gamma$  is the diffusion norm of the corresponding diffusion matrix  $M$ .

*Proof.* During the SOS iterations (Equation 4), we round the flow such that only integer load values are shifted over the edges. Hence, we get

$$w^k = \beta M w^{k-1} - (\beta - 1)w^{k-2} + \delta_{k-1}, \quad (5)$$

where  $\delta_k$  is the round-off error in iteration  $k$ . Comparing this to the balancing algorithm with arbitrary divisible load  $w'^k$  (Equation 4), we get an accumulated round-off  $a_k = w^k - w'^k$  where  $a_0 = 0$  and  $a_1 = \delta_0$ . We know that for any eigenvector  $z_i$  of  $M$  an eigenvector  $u_i$  of  $S^{-1/2}LS^{-1/2}$  exists such that  $z_i = S^{1/2}u_i$ . We also know that the eigenvectors  $z_i$  form a basis in  $\mathbb{R}^n$  [4]. On the other hand,  $a^k = \sum_{i=2}^n \beta_i z_i$  for some  $\beta_2, \dots, \beta_n \in \mathbb{R}$  and any  $k \in \mathbb{N}$ . Combining the techniques from [18] and [4], we therefore obtain

$$\|a_k\|_2 \leq \sum_{j=0}^{k-1} \sqrt{\frac{s_{\max}}{s_{\min}}} (\beta - 1)^{\frac{k-j-1}{2}} (k-j) \delta \leq \sqrt{\frac{s_{\max}}{s_{\min}}} \left( \frac{1 - r^{k+1}}{(1-r)^2} - \frac{(k+1)r^k}{1-r} \right) \delta,$$

where  $\delta$  is chosen such that  $\|\delta_j\|_2 \leq \delta$  for any  $j \in \{0, \dots, k-1\}$  and  $r = \sqrt{\beta - 1}$ . Since  $\delta \leq \sqrt{n} \cdot d$ , the theorem follows.  $\square$

Using similar techniques, the same bound can also be obtained for FOS (without the restrictions). Since  $1/(1 - \gamma) = \mathcal{O}(n^3)$  and  $d \leq n - 1$ , the load discrepancy can at most be  $\mathcal{O}(n^t)$  with  $t$  being small constant.

We now present a randomized strategy, which reduces the final weighted load to an asymptotic optimal value. In the sequel, we only consider the load in  $l_\infty$ -norm, although we can also prove that the algorithm described below achieves an asymptotic optimal balance in  $l_1$ - and  $l_2$ -norm.

In order to show the optimality, we have to consider the lower bound on the final weighted load deviation. Let  $\tilde{w}_i^k = w_i^k / \bar{s}_i$  be the *weighted load* of node  $i$  after  $k$  steps, where  $\bar{s}_i = n \cdot s_i / (\sum_{j=1}^n s_j)$  is its *normalized processing speed*.

**Proposition 1.** There exist node weighted graphs  $G = (V, E)$  and initial load distributions  $w$ , such that for any load balancing scheme working with indivisible unit-size tokens the expected final weighted load in  $l_\infty$ -norm,  $\lim_{k \rightarrow \infty} \|\tilde{w}^k\|$ , is  $\bar{w}_i / \bar{s}_i + \Omega(1)$ .

To see this, let  $G$  be a homogeneous network ( $s_i = 1$  for  $i \in V$ ) and let  $\bar{w}_i = b' + b$ , where  $b' \in \mathbb{N}$  and  $b \in (0, 1)$ . Obviously, this fulfills the proposition.

The following randomized algorithm reduces the final load  $\|\tilde{w}^k\|_\infty$  to  $\bar{w}_i / \bar{s}_i + \mathcal{O}(1)$  in heterogeneous networks. We assume that the weights  $s_i$  are lying in the range  $[1, n^\delta]$  with  $\delta < 1$ , and that the number of tokens  $\sum_{i=1}^n w_i$  is high enough.

Note that the results below can also be obtained if these assumptions do not hold, however, for simplicity we prove the results under these conditions. Although we mainly describe the algorithm for static networks, it can also be used on dynamic networks, obtaining the same bound on the final load deviation.

The algorithm we propose consists of two main phases. In the first phase, we perform FOS as described in [4] to reduce the load imbalance in  $l_1$ -norm to  $\mathcal{O}(n^t)$ , where  $t$  is a small constant value. At the same time, we approximate  $\bar{w}_i$  within an error of  $\pm 1/n$  by using the diffusion algorithm for arbitrarily divisible load. In the second phase, we perform  $\mathcal{O}(\log(n))$  so called *main random walk rounds* (MRWR) described in the following.

In a preparation step, we mark the tokens that take part in the random walk and also introduce participating “negative tokens” that represent load deficiencies. Let  $w_i$  be the current (integer) load and  $R$  be the set of nodes with  $\bar{s}_i < 1/3$ . If a node  $i \in R$  contains less than  $\lfloor \bar{w}_i \rfloor$  tokens, we place  $\lfloor \bar{w}_i \rfloor - w_i$  “negative tokens” on it. If  $i \in R$  owns more than  $\lfloor \bar{w}_i \rfloor$  tokens, we mark  $w_i - \lfloor \bar{w}_i \rfloor$  of them. On a node  $i \in V \setminus R$  with less than  $\lceil \bar{w}_i \rceil + \lceil 2\bar{s}_i \rceil$  tokens, we create  $\lceil \bar{w}_i \rceil + \lceil 2\bar{s}_i \rceil - w_i$  “negative tokens”. Accordingly, if  $i \in V \setminus R$  contains more than  $\lceil \bar{w}_i \rceil + \lceil 2\bar{s}_i \rceil$  tokens, we mark  $w_i - (\lceil \bar{w}_i \rceil + \lceil 2\bar{s}_i \rceil)$  of them. Obviously, the number of “negative tokens” is larger than the number of marked tokens by an additive value of  $\Omega(n)$ . Now, in each MRWR all “negative tokens” and all marked tokens perform a random walk of length  $(ad/\lambda_2) \ln(n)$  according to  $M$ , where  $a$  is the same constant as in Lemma 2. Note, that if a “negative token” is moved from node  $i$  to node  $j$ , a real token is transferred in the opposite direction from node  $j$  to node  $i$ . Furthermore, if a “negative token” and a marked token meet on a node, they eliminate each other. We now show that one MRWR reduces the total number of marked tokens that produced the overload in the system by at least a constant factor  $c$ .

From the description of the algorithm immediately follows that the number of iteration steps is  $\mathcal{O}((d/\lambda_2)(\log^2(n) + \log(E)))$ . In the remaining part of this Section we prove the correctness of the algorithm. In contrast to the above description, we assume for the analysis that “negative tokens” and marked tokens do not eliminate each other instantly, but only after completing a full MRWR. This modification simplifies the proof, although the same bounds can also be obtained for the original randomized algorithm.

**Lemma 3.** *Let  $G = (V, E)$ ,  $|V| = n$  be a node weighted graph. Assume that  $qn$  tokens are executing random walks, according to  $G$ 's diffusion matrix, of length  $(ad \ln(n))/\lambda_2$ ,  $q > 32 \ln(n)$ , where  $a$  is the same constant as in Lemma 2. Then, a constant  $c$  exists, such that, after completion of the random walk, the number of tokens producing overload is less than  $cn\sqrt{q \ln(n)}$  with probability  $1 - o(1/n)$ .*

**Lemma 4.** *Let  $G = (V, E)$ ,  $|V| = n$  be a node weighted graph. Assume that  $qn$  tokens are executing random walks, according to  $G$ 's diffusion matrix, of length  $(ad \ln(n))/\lambda_2$ , where  $a$  is the same constant as in Lemma 2. If  $q$  is larger than some certain constant, then, a constant  $c > 4$  exists, such that, after completion of the random walk, the number of tokens producing overload is less than  $qn/c$  with probability  $1 - o(1/n)$ .*



The proofs of the according Lemmas for homogeneous topologies can be found in [8]. Using some additional techniques we can generalize them to heterogeneous graphs. Due to space limitations, we omit them here.

**Lemma 5.** *Let  $G = (V, E)$ ,  $|V| = n$  be a node weighted graph and let  $q_1 n$  be the number of marked tokens and  $q_2 n$  be the number of “negative tokens” which perform random walks of length  $(\text{ad } \ln(n))/\lambda_2$  on  $G$ , respectively. If  $q_1, q_2 = O(1)$  with  $q_2 n - q_1 n = \Omega(n)$  and  $q_1 > \ln(n)/n$ , then a constant  $c > 1$  exists, such that, after completion of the random walk, the number of tokens producing overload is less than  $\lceil q_1 n/c \rceil$  with probability  $1 - o(1/n)$ .*

*Proof.* First, we describe the distribution of the “negative tokens” on  $G$ ’s nodes. Obviously, with probability at least  $1/2$  a token is placed on a node  $i$  with  $\bar{s}_i > 1/2$ . Using the Chernoff bound [19], it can be shown that with probability  $1 - o(1/n^2)$  at least  $q_2 n(1 - o(1))/2$  tokens are situated on nodes with  $\bar{s}_i > 1/2$  after a MRWR. With help of the Chernoff bound we can also show that for any node  $i$  with  $\bar{s}_i \geq c' \ln(n)$  and  $c'$  being a proper constant, with probability  $1 - o(1/n^2)$  there are at least  $q_2 \bar{s}_i/2$  tokens on  $i$  after the MRWR. If  $G$  contains less than  $c' \ln(n)$  nodes with normalized processing speed  $c' \ln(n)/2^j < \bar{s}_i < c' \ln(n)/2^{j-1}$  for some  $j \in \{1, \dots, \log(c' \ln(n)) + 1\}$ , then a token lies on one of these nodes with probability  $\mathcal{O}(\ln^2(n)/n)$ . Otherwise, if there are  $Q > c' \ln(n)$  nodes with speed  $c' \ln(n)/2^j < \bar{s}_i < c' \ln(n)/2^{j-1}$ , then at least  $Q/\rho_j$  of these nodes contain more than  $\lceil q_2 c' \ln(n)/2^{j+1} \rceil$  tokens, where  $\rho_j = \max\{\lceil 2/(q_2 c' \ln(n)/2^j) \rceil, 2\}$ .

Now we turn our attention to the distribution of the marked tokens after an MRWR. Let  $q_1 n \geq \sqrt{n}$ . Certainly, w.h.p.  $q_1 n(1 - o(1))/2$  tokens reside on nodes with  $\bar{s}_i > 1/2$ . On a node  $i$  with normalized processing speed  $\bar{s}_i > c' \ln(n)$  are placed at most  $q_1 \bar{s}_i(1 + o(1)) + \mathcal{O}(\ln(n))$  tokens. Therefore, most of the marked tokens on such heavy nodes will be eliminated by the “negative tokens”. Let  $S_j$  be the set of nodes with  $c' \ln(n)/2^j < \bar{s}_i < c' \ln(n)/2^{j-1}$ , where  $j \in \{1, \dots, \log(c' \ln(n))\}$ . We ignore the tokens in sets  $S_j$  with  $|S_j| \leq \sqrt{n}$ , since their number adds up to at most  $\mathcal{O}(\sqrt{q_1 n} \ln^2(n))$ . Each of the other sets  $S_j$  with  $|S_j| > \sqrt{n}$ , contains  $n_S = \Omega(\sqrt{q_1 n})$  marked tokens distributed nearly evenly. Therefore, a constant  $c''$  exists so that at least  $n_S/c''$  of them are eliminated by “negative tokens”. If  $q_1 n \leq \sqrt{n}$ , then after the MRWR it holds that a marked token lies on some node  $i$  with normalized speed  $\bar{s}_i > 1/2$  with a probability of at least  $1/2$ . If one of these tokens is situated on some node  $i$  with  $\bar{s}_i \geq c' \ln(n)$ , then it is destroyed by some “negative token” with high probability. If a marked token resides on some node of the sets  $S_j$ , where  $j \in \{1, \dots, \log(c' \ln(n))\}$ , then a constant  $c_f > 0$  exists such that this token is destroyed with probability  $1/c_f$ . Therefore, a marked token is destroyed with probability of at least  $1/(2c_f)$ .

Summarizing, the number of marked tokens is reduced to  $\sqrt{n}$  after  $\mathcal{O}(\ln(n))$  MRWRs. Additional  $\mathcal{O}(\ln(n))$  MRWRs decrease this number to at most  $\mathcal{O}(\ln(n))$ . These can be eliminated within another  $\mathcal{O}(\ln(n))$  MRWRs with probability  $1 - o(1/n)$ .  $\square$

Combining the Lemmas 3, 4, and 5 we obtain the following theorem.



**Theorem 2.** *Let  $G = (V, E)$  be a node weighted graph with maximum vertex degree  $d$  and let  $\lambda_2$  be the second smallest eigenvalue of the weighted Laplacian of  $G$ . Furthermore, let  $w^0$  be the initial load and  $E = \max_i |w_i^0 - \bar{w}_i|$  the initial maximal load imbalance. If  $\sum_{i=1}^n \bar{w}_i$  exceeds some certain threshold, then the randomized algorithm reduces the weighted load in  $l_\infty$ -norm,  $\|\tilde{w}^k\|_\infty$  to  $\bar{w}_i/\bar{s}_i + \mathcal{O}(1)$  in  $k = \mathcal{O}((d/\lambda_2)(\log(E) + \log^2(n)))$  iteration steps with probability  $1 - o(1/n)$ .*

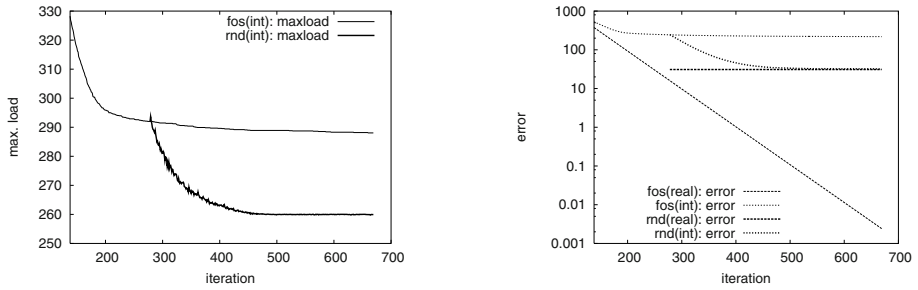
In the previous analysis we assume that the tokens are evenly distributed on the graphs' nodes after each MRWR. However, after the marked and “negative tokens” have eliminated each other, the distribution of the remaining entities no longer corresponds to the stationary distribution of the Markov process. For the analysis, we therefore perform another MRWR to regain an even distribution. Indeed, the tokens are not evenly distributed after the elimination step, but they are close to the stationary distribution in most cases. Hence, the analysis is not tight what can also be seen in the experiments presented in the next Section. Using the techniques of [20], we could improve the upper bound for the runtime of the algorithm for special graph classes. However, we do not see how to obtain better run-time bounds in the general case.

Similar results as presented can also be obtained w.r.t. the  $l_1$  and  $l_2$ -norm. Moreover, the results can also be generalized to dynamic networks. Dynamic networks can be modeled by a sequence  $(G_i)_{i \geq 0}$  of graphs, where  $G_i$  is the graph which occurs in iteration step  $i$  [13]. Let us assume that any graph  $G_i$  is connected and let  $h \geq 1/C \sum_{i=kC+1}^{(k+1)C} (d_{\max}^i/\lambda_2^i)$  for any  $k \in \mathbb{N}$ , where  $C$  is a large constant,  $d_{\max}^i$  represents the maximum vertex degree and  $\lambda_2^i$  denotes the second smallest eigenvalue of  $G_i$ 's Laplacian. Then, the randomized algorithm reduces  $\|\tilde{w}^k\|_\infty$  to  $\bar{w}_i/\bar{s}_i + \mathcal{O}(1)$  in  $k = \mathcal{O}(h(\log(E) + \log^2(n)))$  iteration steps (with probability  $1 - o(1/n)$ ).

## 4 Experiments

In this section we present some of the experimental results we obtained with our implementation of the approach proposed in the last Section. Our tests are performed in two different environments. The first one is based on common network topologies and therefore mainly considers static settings. However, by letting each graph edge only be present with a certain probability  $p$  in each iteration, it is possible to simulate e.g. edge failures and therefore dynamics. The second environment is taken from the mobile ad-hoc network community. Here, nodes move around in a simulation space and a link between two nodes is present if their distance is small enough.

The algorithm works as described in Section 3 and consists of two main phases. First, we balance the tokens as described in [2] only sending the integer amount of the calculated flow. Simultaneously and independently, we simulate the diffusion algorithm with arbitrarily divisible load. Note, that in both cases we adapt the vertex degree in non static networks in every iteration to improve the



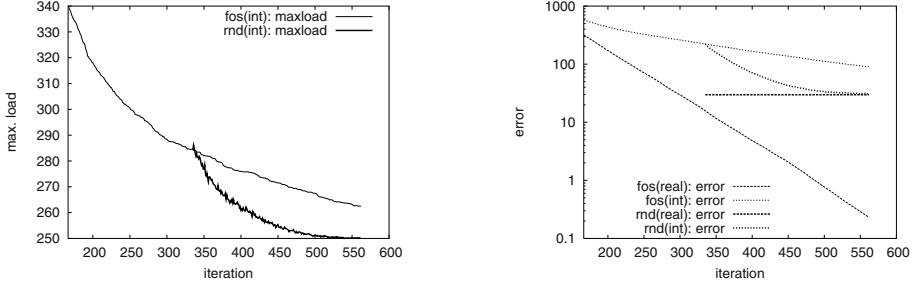
**Fig. 1.** Balancing tokens on the  $16 \times 16$  Torus.

convergence as shown in [6]. The additional simulation leads to an approximation of the fully balanced load amount. If its error is small enough, the second phase of the algorithm is entered. Each node now calculates its desired load based on its approximated and its current real integer load situation as it is described in Section 3. Then, the excess load units and the virtual negative tokens start performing a random walk until the error between the desired load and the real load becomes small enough.

The network types included in our tests are general interconnection topologies like Butterfly, Cube Connected Cycle, de Bruijn, Hypercube, Shuffle-Exchange, Grid, and Torus networks. All of these topologies are well known and some of them are designed to have many favorable properties for distributed computing, e.g. a small vertex degree and diameter and large connectivity.

Figure 1 demonstrates the algorithm's behavior on the  $16 \times 16$  Torus. In this setting, all  $16^4$  token are placed on a single node and an edge fails with 10% probability. Furthermore, the nodes processing speed varies from 0.8 to 1.2. Applying FOS, the maximal weighted load (left) is initially reduced quickly, while in the end many iterations for only small improvements are needed. At some point, due to the rounding, no further reduction will be possible. This is also visible when looking at the  $l_2$  error (right) which stagnates at around 200.0. In contrast, the error of the arbitrarily divisible load simulation converges steadily. Hence, the proposed load balancing algorithm switches to random walk in iteration 277. One can see (left) that this accelerates the load distribution and leads also to a smaller maximal weighted load than FOS can ever achieve. Note, that the optimal load for each node is not exactly known when switching. Hence, the error of the unit-size token distribution cannot exceed the one of the approximated solution (right).

Detailed presentations of experiments based on different graphs classes and with varying parameters have to be omitted due to space limitations. Nevertheless, the results are very similar to the given example, even for graphs that are very well suited for diffusion like the Hypercube. In general, increasing the edge failure probability slows down the algorithm. However, it slightly improves the balance that can be achieved with diffusion, because the average vertex degree is smaller and therefore the rounding error decreases. Furthermore, we observe that the number of additional iterations needed is much smaller than expected re-



**Fig. 2.** Balancing tokens in a mobile ad-hoc network.

garding the theoretical increase of the bound through the additional logarithmic factor.

The second environment we use to simulate load balancing is a mobile ad-hoc network (Manet) model. The simulation area is the unit-square and 256 nodes are placed randomly within it. Prior to an iteration step of the general diffusion scheme, edges are created between nodes depending on their distance. Here, we apply the disc graph model (e.g. [21]) which simply uses a uniform communication radius for all nodes. After executing one load balancing step, all nodes move toward their randomly chosen way point. Once they have reached it, they pause for some iterations before continuing to their next randomly chosen destination. This model has been proposed in [22] and is widely applied in the ad-hoc network community to simulate movement. Note, that when determining neighbors as well as during the movement, the unit-square is considered to have wrap-around borders, meaning that nodes leaving on one side of the square will reappear at the proper position on the other side. For the experiments, we average the results from 25 independent runs.

The outcome of one experiment is shown in Figure 2. The nodes move quite slowly with speed between 0.001 and 0.005, pause for 3 iterations when they have reached their destination, and their communication radius is set to 0.1. In contrast to the former results, the load can be better balanced when applying FOS only. This is due to the vertex movement which is an additional, and in later iterations the only mean to spread the tokens in the system. Higher movement rates will increase this effect. Nevertheless, the randomized algorithm is again able to speed up the load balancing process reaching an almost equalized state much earlier.

## 5 Conclusion

The presented randomized algorithm balances unit-size tokens in general heterogeneous and dynamic networks without global knowledge. It is able to reduce the weighted maximal overload in the system to  $\mathcal{O}(1)$  with only slightly increasing the run-time by at most a factor of  $\mathcal{O}(\ln(n))$  compared to the general diffusion scheme. From our experiments, we can see that this additional factor is usually not needed in practice.

## References

1. G. Cybenko. Load balancing for distributed memory multiprocessors. *J. Par. Dist. Comp.*, 7:279–301, 1989.
2. S. Muthukrishnan, B. Ghosh, and M. H. Schultz. First and second order diffusive methods for rapid, coarse, distributed load balancing. *Theo. Comp. Syst.*, 31:331–354, 1998.
3. R. Diekmann, A. Frommer, and B. Monien. Efficient schemes for nearest neighbor load balancing. *Par. Comp.*, 25(7):789–812, 1999.
4. R. Elsässer, B. Monien, and R. Preis. Diffusion schemes for load balancing on heterogeneous networks. *Theo. Comp. Syst.*, 35:305–320, 2002.
5. V. S. Sunderarm and G. A. Geist. Heterogeneous parallel and distributed computing. *Par. Comp.*, 25:1699–1721, 1999.
6. R. Elsässer, B. Monien, and S. Schamberger. Load balancing in dynamic networks. In *I-SPAN*, 2004.
7. Y. Rabbani, A. Sinclair, and R. Wanka. Local divergence of markov chains and the analysis of iterative load-balancing schemes. In *FOCS*, pages 694–703, 1998.
8. R. Elsässer and B. Monien. Load balancing of unit size tokens and expansion properties of graphs. In *SPAA*, pages 266–273, 2003.
9. A. Cortes, A. Ripoll, F. Cedo, M. A. Senar, and E. Luque. An asynchronous and iterative load balancing algorithm for discrete load model. *J. Par. Dist. Comp.*, 62:1729–1746, 2002.
10. J. E. Gehrke, C. G. Plaxton, and R. Rajaraman. Rapid convergence of a local load balancing algorithm for asynchronous rings. *Theo. Comp. Sci.*, 220:247–265, 1999.
11. M. E. Houle, E. Tempero, and G. Turner. Optimal dimension-exchange token distribution on complete binary trees. *Theo. Comp. Sci.*, 220:363–376, 1999.
12. M. E. Houle, A. Symvonis, and D. R. Wood. Dimension-exchange algorithms for load balancing on trees. In *Sirocco*, pages 181–196, 2002.
13. B. Ghosh, F. T. Leighton, B. M. Maggs, S. Muthukrishnan, C. G. Plaxton, R. Rajaraman, A. Richa, R. E. Tarjan, and D. Zuckerman. Tight analyses of two local load balancing algorithms. *SIAM J. Computing*, 29:29–64, 1999.
14. D. M. Cvetkovic, M. Doob, and H. Sachs. *Spectra of Graphs*. Johann Ambrosius Barth, 3rd edition, 1995.
15. B. Ghosh, S. Muthukrishnan, and M. H. Schultz. First and second order diffusive methods for rapid, coarse, distributed load balancing. In *SPAA*, pages 72–81, 1996.
16. J. A. Fill. Eigenvalue bounds on convergence to stationarity for nonreversible markov chains, with an application to the exclusion process. *Ann. Appl. Probab.*, 1:62–87, 1991.
17. M. Mihail. Conductance and convergence of markov chains: a combinatorial treatment of expanders. In *FOCS*, pages 526–531, 1989.
18. G. Golub. Bounds for the round-off errors in the richardson second order method. *BIT*, 2:212–223, 1962.
19. T. Hagerup and C. Rüb. A guided tour of chernoff bounds. *Inform. Proc. Letters*, 36(6):305–308, 1990.
20. P. Sanders. Analysis of nearest neighbor load balancing algorithms for random loads. *Par. Comp.*, 25:1013–1033, 1999.
21. Y. Wang and X.-Y. Li. Geometric spanners for wireless ad hoc networks. In *ICDCS*, pages 171–180. IEEE, 2002.
22. D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

# Comparing Real Algebraic Numbers of Small Degree

Ioannis Z. Emiris and Elias P. Tsigaridas

Department of Informatics and Telecommunications  
National Kapodistrian University of Athens, Greece  
`{emiris,et}@di.uoa.gr`

**Abstract.** We study polynomials of degree up to 4 over the rationals or a computable real subfield. Our motivation comes from the need to evaluate predicates in nonlinear computational geometry efficiently and exactly. We show a new method to compare real algebraic numbers by precomputing generalized Sturm sequences, thus avoiding iterative methods; the method, moreover handles all degenerate cases. Our first contribution is the determination of rational isolating points, as functions of the coefficients, between any pair of real roots. Our second contribution is to exploit invariants and Bezoutian subexpressions in writing the sequences, in order to reduce bit complexity. The degree of the tested quantities in the input coefficients is optimal for degree up to 3, and for degree 4 in certain cases. Our methods readily apply to real solving of pairs of quadratic equations, and to sign determination of polynomials over algebraic numbers of degree up to 4. Our third contribution is an implementation in a new module of library SYNAPS v2.1. It improves significantly upon the efficiency of certain publicly available implementations: Rioboo's approach on AXIOM, the package of Guibas-Karavelas-Russel [11], and CORE v1.6, MAPLE v9, and SYNAPS v2.0. Some existing limited tests had shown that it is faster than commercial library LEDA v4.5 for quadratic algebraic numbers.

## 1 Introduction

Our motivation comes from computer-aided geometric design and nonlinear computational geometry, where predicates rely on real algebraic numbers of small degree. These are crucial in software libraries such as ESOLID [15], EXACUS (eg. [12], [2]), and CGAL (eg. [8]). Predicates must be decided exactly in all cases, including degeneracies. We focus on real algebraic numbers of degree up to 4 and polynomials in one variable of arbitrary degree or in 2 variables of degree  $\leq 2$ . Efficiency is critical because comparisons on such numbers lie in the inner loop of most algorithms, including those for computing the arrangement of algebraic curves, arcs or surfaces, the Voronoi diagrams of curved objects, eg. [2,7,12,14] and kinetic data-structures [11].

Our work is also a special-purpose quantifier elimination method for one or two variables and for parametric polynomial equalities and inequalities of

low degree. Our approach extends [16,22] because our rational isolating points eliminate the need of multiple sign evaluations in determining the sign of a univariate polynomial over an algebraic number of degree  $\leq 4$ . We also extend the existing approaches so as to solve some simple bivariate problems (sec. 7).

Our method is based on pre-computed generalized Static sequences; in other words, we implement straight-line programs for each comparison. Finding isolating points of low algebraic degree (and rational for polynomials of degree  $\leq 4$ ) is a problem of independent interest. It provides starting points for iterative algorithms and has direct applications, e.g. [7]. Our Sturm-based algorithms rely on isolating points in order to avoid iterative methods (which depend on separation bounds) and the explosion of the algebraic degree of the tested quantities. In order to reduce the computational effort, we factorize the various quantities by the use of invariants and/or by the elements of the Bezoutian matrix; for our implementation, this is done in an automated way.

We have implemented a package of algebraic numbers as part of the SYNAPS 2.1 library [6] and show that it compares favorably with other software. Our code can also exist as a stand-alone C++ software package. We call our implementation  $S^3$  which stands for *Static Sturm Sequences* (or *Salmon-Sturm-Sylvester*).

The following section overviews some of the most relevant existing work. Next, we formalize Sturm sequences. Sect. 4 studies discrimination systems and its connection to the invariants of the polynomial. Sect. 5 obtains rational isolating points for degree  $\leq 4$ . Sect. 6 bounds complexity and sect. 7 applies our tools to sign determination and real solving and indicates some of our techniques for automatic code generation. Sect. 8 illustrates our implementation with experimental results. We conclude with future work.

## 2 Previous Work and Contribution

Although the roots of rational polynomials of degree up to 4 can be expressed explicitly with radicals, the computation of the real roots requires square and cubic roots of complex numbers. Even if only the smallest (or largest) root is needed, one has to compute all real roots (cf [13]). Another critical issue is that there is no formula that provides isolating rational points between the real roots of polynomials: this problem is solved in this paper for degree  $\leq 4$ .

In quantifier elimination, an effort was made to optimize low level, operations, eg. [16,22]. However, by that approach, there is a need for multiple Sturm sequences. By our approach, we need to evaluate only one Sturm sequence in order to decide the sign of a polynomial over a cubic or quartic algebraic number.

Our discrimination system for the quartic is the same as that in [23], but is derived differently and corrects a small error in [23].

Rioboo implemented in AXIOM an arithmetic of real algebraic numbers of arbitrary degree with coefficients from a real closed field [17]. The extension he proposed for the sign evaluation is essentially based upon theorem 2.

**Table 1.** Computing the arrangement of 300 circular arcs with: random full circles (rfc), circles centered on the crosspoints and the centers of the cells of a square grid of cell size  $10^4$  (g), the same perturbed (pg), random monotone arcs (rma), random non-monotone arcs (rnma). All the geometry code was done in CGAL and so the first column shows the arithmetic used.

sec	rfc	g	pg	rma	rnma
$S^3$	137	2	5	10	34
CORE v1.6	—	10	17	20	195
LEDA v4.5	374	5	11	19	104

Iterative methods based on the approach of Descartes / Uspensky seem to be the fastest means of isolating real roots, in general (cf. [18]). Such methods are implemented in SYNAPS. An iterative method using Sturm sequences, has been implemented in [11]. Both methods are tested in sec. 8.

LEDA and CORE<sup>1</sup> evaluate expression trees built recursively from integer operations and  $\sqrt{\phantom{x}}$ , and rely on separation bounds. LEDA treats arbitrary algebraic numbers, by the *diamond operator*, based on Descartes/Uspensky iteration. But it faces efficiency problems ([20]) in computing isolating intervals for degree 3 and 4, since Newton’s iteration cannot always be applied with interval coefficients. CORE recently provided for dealing with algebraic numbers using Sturm sequences. Currently this operator cannot handle multiple roots.

Precomputed quantities for the comparison of quadratic algebraic numbers were used in [5], derived from the *u*-resultant and Descartes’ rule of sign. In [14], the same problem was solved with static Sturm sequences, thus improving upon the runtime by up to 10%. In generalizing these methods to higher degree, it is not obvious how to determine the (invariant) quantities to be tested in order to minimize the bit complexity. Another major issue is the isolating points as well as the need of several Sturm sequences.

The contribution of this paper starts with quadratic numbers, for which we reformulated the existing method in order to make it generalizable to higher degree [10]. The efficiency of our implementation for quadratic numbers is illustrated in [8]; we copy table 1 from this paper. For algebraic numbers of degree 3 and 4, preliminary results are in [10,9], where more details can be found, which cannot fit here for reasons of space. Our novelty is to use a *single* Sturm sequence for degree up to 4 (see cor. 4), first by considering the discrimination system of the given polynomial for purposes of root classification and in order to derive a square-free polynomial defining the algebraic number, second by deriving *rational* isolating points (theorems 6 and 10) and finally by reducing the computational effort through factoring of the tested quantities using invariants and elements of the Bezoutian matrix. Our implementation computes these quantities in an automatic way.

<sup>1</sup> <http://www.algorithmic-solutions.com/enleda.htm>,  
<http://www.cs.nyu.edu/exact/core>

### 3 Sturm Sequences

Sturm sequences is a well known and useful tool for isolating the roots of any polynomial (cf [24], [1]). Additionally, the reader can refer to [14] where Sturm sequences are used for comparing algebraic numbers of degree 2, or to [5] where the comparison of such numbers was done by the resultant. In the sequel  $\mathbf{D}$  is a ring,  $\mathbf{Q}$  is its fraction field and  $\overline{\mathbf{Q}}$  the algebraic closure of  $\mathbf{Q}$ . Typically  $\mathbf{D} = \mathbb{Z}$  and  $\mathbf{Q} = \mathbb{Q}$ .

**Definition 1.** Let  $P$  and  $Q \in \mathbf{D}[x]$  be nonzero polynomials. By a (generalized) Sturm sequence for  $P, Q$  we mean any pseudo-remainder sequence  $\overline{P} = (P_0, P_1, \dots, P_n)$ ,  $n \geq 1$ , such that for all  $i = 1, \dots, n$ , we have  $a_i P_{i-1} = Q_i P_i + b_i P_{i+1}$  ( $Q_i \in \mathbf{D}[x]$ ,  $a_i, b_i \in \mathbf{D}$ ), such that  $a_i b_i < 0$  and  $P_0 = P, P_1 = Q, P_{n+1} = 0$ . We usually write  $\overline{P}_{P_0, P_1}$  if we want to indicate the first two terms in the sequence.

For a Sturm sequence  $\overline{P}$ ,  $V_{\overline{P}}(p)$  denotes the number of sign variations of the evaluation of the sequence at  $p$ . The last polynomial in  $\overline{P}_{P_0, P_1}$  is the resultant of  $P_0$  and  $P_1$ .

**Theorem 2.** Let  $P, Q \in \mathbf{D}[x]$  be relatively prime polynomials and  $P$  square-free. If  $a < b$  are both non-roots of  $P$  and  $\gamma$  ranges over the roots of  $P$  in  $[a, b]$ , then

$$V_{P, Q}[a, b] := V_{P, Q}(a) - V_{P, Q}(b) = \sum_{\gamma} \text{sign}(P'(\gamma)Q(\gamma)).$$

where  $P'$  is the derivative of  $P$ .

**Corollary 3.** Theorem 2 holds if in place of  $Q$  we use  $R = \text{PRem}(Q, P)$ , where  $\text{PRem}(Q, P)$ , stands for the pseudo-remainder of  $Q$  divided by  $P$ .

Finding isolating intervals for the roots of any polynomial is done below. Still, the computation of a Sturm sequence is quite expensive. In order to accelerate the computation we assume that the polynomials are  $P, Q \in \mathbf{D}[a_0, \dots, a_n, b_0, \dots, b_m][x]$ , where  $a_i, b_j$  are the coefficients considered as parameters, and we pre-compute various Sturm sequences ( $n, m \leq 4$ ).

The isolating-interval representation of real algebraic number  $\alpha \in \overline{\mathbf{Q}}$  is  $\alpha \cong (A(X), I)$ , where  $A(X) \in \mathbf{D}[X]$  is square-free and  $A(\alpha) = 0$ ,  $I = [a, b]$ ,  $a, b \in \mathbf{Q}$  and  $A$  has no other root in  $I$ .

**Corollary 4.** Assume  $B(X) \in \mathbf{D}[X] : \beta = B(\alpha)$ , and that  $\alpha \cong (A, [a, b])$ . By theorem 2,  $\text{sign}(B(\alpha)) = \text{sign}(V_{A, B}[a, b] \cdot A'(\alpha))$ .

Let us compare two algebraic numbers  $\gamma_1 \cong (P_1(x), I_1)$  and  $\gamma_2 \cong (P_2(x), I_2)$  where  $I_1 = [a_1, b_1]$  and  $I_2 = [a_2, b_2]$ . Let  $J = I_1 \cap I_2$ . When  $J = \emptyset$ , or only one of  $\gamma_1$  and  $\gamma_2$  belong to  $J$ , we can easily order the 2 algebraic numbers. All these tests are implemented by the previous corollary and theorem. If  $\gamma_1, \gamma_2 \in J$ , then  $\gamma_1 \geq \gamma_2 \Leftrightarrow P_2(\gamma_1) \cdot P_2'(\gamma_2) \geq 0$ . We can easily obtain the sign of  $P_2(\gamma_2)$ , and from theorem 2, we obtain the sign of  $P_2(\gamma_1)$ .



## 4 Root Classification

Before applying the algorithms for root comparison, we analyze each polynomial by determining the number and the multiplicities of its real roots. For this, we use a system of discriminants. For the quadratic polynomial the discrimination system is trivial. For the cubic, it is well known [22,10]. We study the quartic by Sturm-Habicht sequences, while [23] used a resultant-like matrix. For background see [24,1]. We factorize the tested quantities by invariants and elements of the Bezoutian matrix. We use invariants in order to provide square-free polynomials defining the algebraic numbers, to compute the algebraic numbers as rationals if this is possible and finally to provide isolating rationals.

Consider the quartic polynomial equation, where  $a > 0$ .

$$f(X) = aX^4 - 4bX^3 + 6cX^2 - 4dX + e. \quad (1)$$

In the entire paper, we consider as input the coefficients  $a, b, c, d, e \in \mathbf{D}$ .

For background on invariants see [4], [19]. We consider the rational invariants of  $f$ , i.e the invariants in  $GL(2, \mathbb{Q})$ . They form a graded ring [4], generated by:

$$A = W_3 + 3\Delta_3, \quad B = -dW_1 - e\Delta_2 - c\Delta_3. \quad (2)$$

Every other invariant is an isobaric polynomial in  $A$  and  $B$ , i.e. it is homogeneous in the coefficients of the quartic. We denote the invariant  $A^3 - 27B^2$  by  $\Delta_1$  and refer to it as the *discriminant*. The semivariants (which are the leading coefficients of the covariants) are  $A, B$  and:

$$\Delta_2 = b^2 - ac, R = aW_1 + 2b\Delta_2, Q = 12\Delta_2^2 - a^2A. \quad (3)$$

We also derived the following quantities, which are not necessarily invariants but they are elements of the Bezoutian matrix of  $f$  and  $f'$ . Recall that the determinant of the Bezoutian matrix equals the resultant, but its size is smaller than the Sylvester matrix [3].

$$\begin{array}{ll} \Delta_3 = c^2 - bd & T = -9W_1^2 + 27\Delta_2\Delta_3 - 3W_3\Delta_2 \\ \Delta_4 = d^2 - ce & T_1 = -W_3\Delta_2 - 3W_1^2 + 9\Delta_2\Delta_3 \\ W_1 = ad - bc & T_2 = AW_1 - 9bB \\ W_2 = be - cd & W_3 = ae - bd \end{array} \quad (4)$$

**Proposition 5.** [23] *Let  $f(X)$  be as in (1). The table gives the real roots and their multiplicities. In case (2) there are 4 complex roots, while in case (8) there are 2 complex double roots (In [23] there is a small error in defining  $T$ ).*

(1) $\Delta_1 > 0 \wedge T > 0 \wedge \Delta_2 > 0$	$\{1, 1, 1, 1\}$
(2) $\Delta_1 > 0 \wedge (T \leq 0 \vee \Delta_2 \leq 0)$	$\{\}$
(3) $\Delta_1 < 0$	$\{1, 1\}$
(4) $\Delta_1 = 0 \wedge T > 0$	$\{2, 1, 1\}$
(5) $\Delta_1 = 0 \wedge T < 0$	$\{2\}$
(6) $\Delta_1 = 0 \wedge T = 0 \wedge \Delta_2 > 0 \wedge R = 0$	$\{2, 2\}$
(7) $\Delta_1 = 0 \wedge T = 0 \wedge \Delta_2 > 0 \wedge R \neq 0$	$\{3, 1\}$
(8) $\Delta_1 = 0 \wedge T = 0 \wedge \Delta_2 < 0$	$\{\}$
(9) $\Delta_1 = 0 \wedge T = 0 \wedge \Delta_2 = 0$	$\{4\}$

## 5 Rational Isolating Points

In what follows,  $f \in \mathbb{Z}[X]$  and  $a > 0$ ; the same methods work for any computable real subfield  $\mathbf{D}$ . It is known that for the quadratic  $f(X) = aX^2 - 2bX + c$ , the rational number  $\frac{b}{a}$ , isolates the real roots.

**Theorem 6.** *Consider the cubic  $f(X) = aX^3 - 3bX^2 + 3cX - d$ . The rational numbers  $\frac{b}{a}$  and  $-\frac{W_1}{2\Delta_2}$  isolate the real roots.*

*Proof.* In [10], we derive the rational isolating points based on the fact that the two extreme points and the inflexion point of  $f(X)$  are colinear. Moreover, the line through these points intersects the  $x$ -axis at a rational number. Notice that the algebraic degree of the isolating points is at most 2. Interestingly, the same points are obtained by applying theorem 7.  $\square$

**Theorem 7.** [21] *Given a polynomial  $P(X)$  with adjacent real roots  $\gamma_1, \gamma_2$ , and any two other polynomials  $B(X), C(X)$ , let  $A(X) := B(X)P'(X) + C(X)P(X)$  where  $P'$  is the derivative of  $P$ . Then  $A(X)$  or  $B(X)$  are called isolating polynomials because at least one of them has at least one real root in the closed interval  $[\gamma_1, \gamma_2]$ . In addition, it is always possible to have  $\deg A + \deg B \leq \deg P - 1$ .*

We now study the case of the quartic. By theorem 7 it is clear how to isolate the roots by 2 quadratic algebraic numbers and a rational. In order to obtain an isolating polynomial, let  $B(X) = ax - b$  and  $C(X) = -4a$  then

$$A(X) = 3\Delta_2 X^2 + 3W_1 X - W_3. \quad (5)$$

Since  $\frac{b}{a}$  is the arithmetic mean of the 4 roots, it is certainly somewhere between the roots. The other two isolating points are the solutions of (5), i.e

$$\sigma_{1,2} = \frac{-3W_1 \pm \sqrt{9W_1^2 + 12\Delta_2 W_3}}{6\Delta_2}. \quad (6)$$

We verify that  $\text{sign}\left(f\left(\frac{b}{a}\right)\right) = \text{sign}\left(a^2 A - 3\Delta_2^2\right)$ , so

$$\begin{cases} \sigma_1 < \frac{b}{a} < \sigma_2, & \text{if } f\left(\frac{b}{a}\right) > 0; \\ \sigma_1 < \sigma_2 < \frac{b}{a}, & \text{if } f\left(\frac{b}{a}\right) < 0 \wedge R > 0; \\ \frac{b}{a} < \sigma_1 < \sigma_2, & \text{if } f\left(\frac{b}{a}\right) < 0 \wedge R < 0; \end{cases} \quad (7)$$

where  $R$  is from (3). If  $f\left(\frac{b}{a}\right) = 0$  then we know exactly one root and can express the other three roots as roots of a cubic. To obtain another isolating polynomial, we use  $B(X) = dx - e, C(X) = -4d$ , and

$$A(X) = W_3 X^3 - 3W_2 X^2 - 3\Delta_4 X.$$

By the theorem at least 2 of the numbers below separate the roots.

$$0, \tau_{1,2} = \frac{3W_2 \pm \sqrt{9W_2^2 + 12\Delta_4 W_3}}{6W_3}. \quad (8)$$

We assume that the roots are  $> 0$ , so 0 is not an isolating point. The order of the isolating points is determined similarly as in (7).

Let us now find rational isolating points for all relevant cases of prop. 5.

$\{1, 1, 1, 1\}$  Treated below.

$\{1, 1\}$  Since  $\{1, 1, 1, 1\}$  is harder, we do not examine it explicitly.

$\{2, 1, 1\}$  The double root is rational since it is the only root of  $\text{GCD}(f, f')$  and its value is  $\frac{T_1}{T_2}$ , see eq (4). In theory, we could divide it out and use the isolating points of the cubic, but in practice we avoid division. When the double root is the middle root then  $\frac{b}{a}$  and  $-\frac{W_1}{2\Delta_2}$  are isolating points, otherwise we use theorem 7 to find one more isolating point in  $\mathbb{Q}$ .

$\{2\}$  Compute the double root from  $\overline{P}_{f, f'}$ ; it is rational as a root of  $\text{GCD}(f, f')$ .

$\{2, 2\}$  The roots are the smallest and largest root of the derivative i.e. a cubic.

Alternatively, we express them as the roots of  $3\Delta_2 X^2 + 3W_1 X - W_3$ .

$\{3, 1\}$  The triple root is  $-\frac{W_1}{2\Delta_2}$  and the single root is  $\frac{3aW_1 + 8b\Delta_2}{2a\Delta_2}$ .

$\{4\}$  The one real root is  $\frac{b}{a} \in \mathbb{Q}$ .

It remains to consider the case where the quartic has 4 simple real roots. We assume that 0 is not a root (otherwise we deal with a cubic), therefore,  $e \neq 0$ . WLOG, we may consider equation (1) with  $b = 0$ . Then, specialize equations (6) and (8) using  $b = 0$ . The only difficult case is when  $\tau_i$  and  $\sigma_j$ ,  $i, j \in \{1, 2\}$ , isolate the same pair of adjacent roots. WLOG, assume that these are  $\tau_1, \sigma_1$ . We combine them by the following lemma.

**Lemma 8.** For any  $m, n, m', n' \in \mathbb{N}^*$ ,  $0 < \frac{m}{n} < \frac{m'}{n'} \Rightarrow \frac{m}{n} < \frac{m+m'}{n+n'} < \frac{m'}{n'}$ .

$$\mathcal{A} := 9\Delta_4 - 3ce, \quad \mathcal{B} := 12ae\Delta_4 + 9d^2c^2 \quad (9)$$

then, an isolating point is  $\frac{3d-3dc+\sqrt{\mathcal{A}+\sqrt{\mathcal{B}}}}{6c+2ae}$ . If we find an integer  $K \in [\sqrt{\mathcal{A}}, \sqrt{\mathcal{B}}]$ , then it suffices to replace  $\sqrt{\mathcal{A}} + \sqrt{\mathcal{B}}$  by  $2K$  and we denote the resulting rational by  $\sigma_i \oplus \tau_j$ ; notice it has degree 2 in the input coefficients. By prop. 5(1),  $\Delta_2 > 0 \Rightarrow c < 0$ . Descartes' rule implies that, if  $e > 0$ , then there are 2 positive and 2 negative roots, while  $e < 0$  means there are 3 positive and one negative root or vice versa. We set  $K = \lceil \sqrt{\mathcal{A}} \rceil$  to prove theorem 10, provided the following holds:

**Theorem 9.** For every quartic in  $\mathbb{Z}[X]$  with 4 distinct real roots and  $b = 0$ , we have  $\sqrt{\mathcal{B}} - \sqrt{\mathcal{A}} \geq 1$ , using notation (9).

*Proof.*

$$\begin{aligned} \sqrt{\mathcal{B}} \geq 1 + \sqrt{\mathcal{A}} &\Leftrightarrow \sqrt{\frac{\mathcal{B}}{\mathcal{A}}} \geq 1 + \frac{1}{\sqrt{\mathcal{A}}} \Leftrightarrow \sqrt{\frac{\mathcal{B}}{\mathcal{A}}} \geq 2 \Leftrightarrow \\ g &:= 4ad^2 - 4ace^2 + 3d^2c^2 - 12d^2 + 16ce \geq 0. \end{aligned}$$

First we show that the minimum of  $g(a, c, d, e)$  is positive, subject to  $-a \leq 1$ ,  $c \leq -5$ , and  $-e \leq -5$ ; we treat the case where  $c > -5$  and  $e < 5$  later. We introduce slack variables  $y_1, y_2, y_3$  and use Lagrange multipliers. So our problem now is

$$\begin{aligned} \min \quad & L(a, c, d, e, y_1, y_2, y_3, \lambda_1, \lambda_2, \lambda_3) := \\ \min \quad & [g(c, e) + \lambda_1(c + y_1^2 + 5)\lambda_2(-e + y_2^2 + 5) + \lambda_3(-a + y_3^2 + 1)] \end{aligned} \quad (10)$$

We take partial derivatives, equate them to zero and the solution of the system, by MAPLE 9, is  $(a, c, d, e) = (1, -5, 0, 5)$  and  $g(1, -5, 0, 5) = 300 > 0$  which is a local minimum. If  $-5 < c < 0$  and  $0 < e < 5$  we check exhaustively that  $\sqrt{\mathcal{B}} - \sqrt{\mathcal{A}} \geq 1$ . If  $e < 0$  then we use again Lagrange multipliers but with the constraint  $e + 1 - y_2^2$ .  $\square$

**Theorem 10.** *Consider a quartic as in (1). At least three of the rational numbers  $\{0, \frac{b}{a}, \frac{e}{d}, \sigma_i \oplus \tau_j\}$  isolate the real roots,  $i, j \in \{1, 2\}$ .*

## 6 Complexity of Computations

The comparison of the roots of two polynomials of degree  $d \leq 4$  using Sturm sequences and isolating intervals provides the following bounds in the degree of the tested quantities. We measure degree in terms of the input quartics' coefficients. A lower bound is the degree of the resultant coefficients, which is  $2d$  in terms of the input coefficients. Recall that the resultant is an irreducible polynomial in the coefficients of an overconstrained system of  $n + 1$  polynomials in  $n$  variables, the vanishing of which is the minimum condition of solvability of the system ([24], [1]).

There is a straightforward algorithm for the comparison of quadratic algebraic numbers, with maximum algebraic degree 4, hence optimal ([14], [5]).

**Theorem 11.** [10] *There is an algorithm for the comparison of algebraic cubic numbers (including all degenerate cases), with maximum algebraic degree 6, hence optimal.*

**Theorem 12.** *There is an algorithm that compares any two roots of two square-free quartics with algebraic degree 8 or 9, depending on the degree of the isolating points. When the quartics are not square-free, the algebraic degree is between 8 and 13. The algorithm needs at most 172 additions and multiplications in order to decide the order. These bounds cover all degenerate cases, including when one polynomial drops degree.*

*Proof. (Sketch)* In [10] we derive this bound by considering the evaluation of the Sturm sequence polynomials over the isolating points (see the discussion after cor. 4 for additional details). The isolating points have degree at most 2. The length of the sequence is at most 6 and so we need at most 12 polynomial evaluations, hence a fixed number of operations. The number of operations is obtained by MAPLE's function `cost`.  $\square$

## 7 Applications

We have implemented a software package, including a `root_of` class for algebraic numbers of degrees up to 4 as part of library SYNAPS (v2.1) [6]. Some function-

alities pertain to higher degrees. Our implementation is generic in the sense that it can be used with any number type and any polynomial class that supports elementary operations and evaluations. It can handle all degenerate cases and has been extended to arbitrary degree (though without isolating points for now). We developed programs that produce all possible sign combinations of the tested quantities, allow us to test as few quantities as possible, and produce both **C++** code and pseudo-code for the comparison and sign determination functions. We provide the following functionalities, where **UPoly** and **BPoly** stand for arbitrary univariate and quadratic bivariate polynomial respectively:

**Sturm(UPoly  $f_1$ , UPoly  $f_2$ )** Compute the Sturm sequence of  $f_1, f_2$ , by pseudo-remainders, (optimized) subresultants, or Sturm-Habicht.

**compare(root\_of  $\alpha$ , root\_of  $\beta$ )** Compare 2 algebraic numbers of degree  $\leq 4$  using precomputed Sturm sequences. We have precomputed all the possible sign variations so as to be able to decide with the minimum number of tests (and faster than with Sturm-Habicht sequences).

**sign\_at(UPoly  $f$ , root\_of  $\alpha$ )** Determination of the sign of a univariate polynomial of arbitrary degree, over an algebraic number of degree  $\leq 4$ . We use the same techniques, as in **compare**.

**sign\_at(BPoly  $f$ , root\_of  $\gamma_x$ , root\_of  $\gamma_y$ )**  $\gamma_1 \cong (P_1(x), I_1)$  and  $\gamma_2 \cong (P_2(x), I_2)$ , where  $I_1 = [a_1, b_1]$  and  $I_2 = [a_2, b_2]$ , are of degrees  $\leq 4$ . We compute the Sturm-Habicht sequence of  $P_1$  and  $f$  wrt  $X$ . We specialize the sequence for  $X = a_1, X = a_2$ , and find the sign of all  $y$ -polynomials over  $\gamma_y$  as above. The Sturm-Habicht sequence was computed in **MAPLE**, using the elements of the Bezoutian matrix. Additionally we used the packages **codegeneration**, **optimize**, in order to produce efficient **C++** code.

**solve(UPoly  $f$ )** Returns the roots of  $f$  as **root\_of**. If  $f$  has degree  $\leq 4$  we use the discrimination system, otherwise we use non-static Sturm sequences.

**solve(BPoly  $f_1$ , BPoly  $f_2$ )** We consider the resultants  $R_x, R_y$  of  $f_1, f_2$  by eliminating  $y$  and  $x$  respectively, thus obtaining degree-4 polynomials in  $x$  and  $y$ . The isolating points of  $R_x, R_y$  define a grid of boxes, where the intersection points are located. The grid has 1 to 4 rows and 1 to 4 columns. It remains to decide, for boxes, whether they are empty and, if not, whether they contain a simple or multiple root. Multiple roots of the resultants are either rational or quadratic algebraic numbers by prop. 5. Each box contains at most one intersection point. We can decide if a box is empty by 2 calls to the bivariate **sign\_at** function. We can do a significant optimization by noticing that the intersections in a column (row) cannot exceed 2 nor the multiplicity of the algebraic number and thus excluding various boxes.

Unlike [2], where the boxes cannot contain any critical points of the intersecting conics, our algorithm does not make any such assumption, hence there is no need to refine them. Our approach can be extended in order to compute intersection points of bivariate polynomials of arbitrary degree, provided that we obtain isolating points for the roots of the two resultants, either statically (as above) or dynamically.

**Table 2.** Running times in *msec* for comparing specific roots of 2 quartics. In parentheses are the number types: **gmpq** / **gmp mpf** stand for GMP rationals / floating point.

msec	$\mathbb{Z}$	$\mathbb{Q}$	far	M- $\mathbb{Q}$	M	$\mathbb{Z}$	$\mathbb{Q}$	far	M- $\mathbb{Q}$	M
AXIOM	38.0	57.4	77.3	67.2	82.3					
MAPLE 9	33.2	52.4	69.0	75.5	74.7					
CORE	8.41	5.67	6.76	9.31	10.1					
SYNAPS(gmpq)	1.097	0.820	0.596	1.480	2.114	2.687	2.693	2.780	3.764	5.698
[11](gmpq)	1.249	0.921	0.991	1.582	1.544	23.74	64.4	7.3	4.121	54.7
[11]-FILT(gmpq)	0.346	0.301	0.279	0.313	0.320	1.594	2.130	1.035	2.758	3.980
$S^3$ (gmpq)	0.077	0.083	0.082	0.074	0.077	0.117	0.190	0.115	0.161	0.129
SYNAPS(gmp mpf)	0.302	0.202	0.195	0.339	0.385					
$S^3$ (gmp mpf)	0.060	0.064	0.057	0.063	0.061					
SYNAPS(double)	0.055	0.042	0.043	0.061	0.071					
$S^3$ (double)	0.010	0.011	0.012	0.010	0.010					

8 Experimental Results

We performed tests on a 2.6GHz Pentium with 512MB memory. The results are on table 2. AXIOM refers to the implementation of real algebraic arithmetic by Rioboo (current CVS version); it is meant, like MAPLE 9, only for a rough comparison. The package in [11] uses subdivision based on Sturm sequences. Row [11]-FILT is the same but the program tries to decide with **double** arithmetic and, if it cannot, then switches to exact arithmetic.  $S^3$  is our code implemented in SYNAPS 2.1.

Every test is averaged over 10000 polynomial pairs. The left part of the table includes results for polynomials with 4 random real roots between -20 and 20, multiplied by an integer  $\in [1, 20]$ , so the coefficients are integers of absolute value  $\leq 32 \cdot 10^5$  and the Mignotte polynomials are  $a(x^4 - 2(Lx - 1)^2)$ , where  $a, L$  are chosen randomly from  $[3, 30]$ . The right part of the table tests polynomials with random roots  $\in [10 \cdot 10^4, 11 \cdot 10^4]$  and Mignotte polynomials with parameters  $a, L$  both chosen randomly from  $[10 \cdot 10^4, 11 \cdot 10^4]$  with uniform distribution.

Column  $\mathbb{Z}$  indicates polynomials with 4 integer roots. Column  $\mathbb{Q}$  indicates polynomials with 4 rational roots. Column *far* indicates that one polynomial has only negative roots while the other one has only positive ones.  $M - \mathbb{Q}$  indicates that one is a Mignotte polynomial and the other has 4 rational roots.  $M$  indicates that we compare the roots of two Mignotte polynomials.

CORE cannot handle multiple roots. This is also the case for the iterative solver of SYNAPS, which also has problems when the roots are endpoints of a subdivision. MAPLE cannot always handle equality for the roots of two Mignotte polynomials. Of course all the implementations have problems when the number type is not an exact one. This is the case of **double** and **gmp mpf**. By considering the left part of table 2, our code is 103, 15 and 16 times faster than CORE, SYNAPS and [11], respectively, in the average case and when no filtering is used. Even

in case of filtering ([11]-FILT) our code is faster by a factor of 4 on average. Increasing the magnitude of the coefficients or decreasing the separation bound of the roots, leads to even more dramatic improvement in the efficiency of our implementation over the other ones; this is the case for the right part of table 2.

$S^3$  has similar running times for all kinds of tests and handles degenerate cases faster than the general ones since we use the discrimination system. This is not the case for any of the other approaches. The running times of our code with an exact number type is less than 8 times slower than when used with doubles. However the latter does not offer any guarantee for the computed result. This is an indication that exact arithmetic imposes a reasonable overhead on efficiency when used in conjunction with efficient algorithms and carefully implemented.

## 9 Future Work

Consider the quintic  $ax^5 + 10cx^3 - 10dx^2 + 5ex - f$ , with  $a > 0$  and  $c < 0$  by assuming 5 real roots. Using the techniques above, we obtain 2 pairs of isolating polynomials. Two of them are

$$B_1 = 2X^2ca + 3dXa + 8c^2 \quad B_2 = (-4df + 4e^2)X^2 + feX - f^2 \quad (11)$$

One pair of roots may be separated by the roots of  $B_1, B_2$ . We combine them by finding an integer between the roots of the respective discriminants:  $\Delta_{B_1} = a(9d^2a - 64c^3)$ ,  $\Delta_{B_2} = (17e^2 - 16df)f^2$ . It suffices to set  $K$  to  $2\lceil\Delta_{B_1}\rceil$  or  $2\lceil\Delta_{B_2}\rceil$ , depending on the signs of  $e, f$ . Tests with MAPLE support our choices.

Filtering should lead to running times similar to those of the double number type. Additionally we are planning to compare our software against the software of [18] and NIX, the polynomial library of EXACUS.

The algebraic degree of the resultant is a tight lower bound in checking solvability, but is it tight for comparisons?

**Acknowledgments.** Both authors acknowledge inspirational comments by R. Rioboo, partial support by INRIA's project "CALAMATA", a bilateral collaboration between the GALAAD group of INRIA Sophia-Antipolis (France) and the National Kapodistrian University of Athens, and by PYTHAGORAS, a project under the EPEAEK program of the Greek Ministry of Education and Religions.

## References

1. S. Basu, R. Pollack, and M-F.Roy. *Algorithms in Real Algebraic Geometry*, volume 10 of *Algorithms and Computation in Mathematics*. Springer-Verlag, 2003.
2. E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schomer. A computational basis for conic arcs and boolean operations on conic polygons. In *ESA*, volume 2461 of *LNCS*, pages 174–186. Springer-Verlag, 2002.
3. P. Bikker and A. Y. Uteshev. On the Bézout construction of the resultant. *J. Symbolic Computation*, 28(1–2):45–88, July/Aug. 1999.

4. J. E. Cremona. Reduction of binary cubic and quartic forms. *LMS J. Computation and Mathematics*, 2:62–92, 1999.
5. O. Deviller, A. Fronville, B. Mourrain, and M. Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. *Comp. Geom. Theory & Appl., Spec. Issue*, 22:119–142, 2002.
6. G. Dos Reis, B. Mourrain, R. Rouillier, and P. Trébuchet. An environment for symbolic and numeric computation. In *Proc. of the International Conference on Mathematical Software 2002*, World Scientific, pages 239–249, 2002.
7. L. Dupont, D. Lazard, S. Lazard, and S. Petitjean. Near-optimal parameterization of the intersection of quadrics. In *Proc. Annual ACM Symp. on Comp. Geometry*, pages 246–255. ACM, June 2003.
8. I. Emiris, A. Kakargias, M. Teillaud, E. Tsigaridas, and S. Pion. Towards an open curved kernel. In *Proc. Annual ACM Symp. on Computational Geometry*, pages 438–446, New York, 2004. ACM Press.
9. I. Z. Emiris and E. P. Tsigaridas. Comparison of fourth-degree algebraic numbers and applications to geometric predicates. Tech. Rep ECG-TR-302206-03, INRIA Sophia-Antipolis, 2003.
10. I. Z. Emiris and E. P. Tsigaridas. Methods to compare real roots of polynomials of small degree. Tech. Rep ECG-TR-242200-01, INRIA Sophia-Antipolis, 2003.
11. L. Guibas, M. Karavelas, and D. Russel. A computational framework for handling motion. In *Proc. 6th Workshop (ALENEX)*, Jan. 2004. To appear.
12. M. Hemmer, E. Schömer, and N. Wolpert. Computing a 3-dimensional cell in an arrangement of quadrics: Exactly and actually! In *Proc. Annual ACM Symp. Comput. Geometry*, pages 264–273, 2001.
13. D. Kaplan and J. White. Polynomial equations and circulant matrices. *The Mathematical Association of America (Monthly)*, 108:821–840, November 2001.
14. M. Karavelas and I. Emiris. Root comparison techniques applied to the planar additively weighted Voronoi diagram. In *Proc. Symp. on Discrete Algorithms (SODA-03)*, pages 320–329, Jan. 2003.
15. J. Keyser, T. Culver, D. Manocha, and S. Krishnan. ESOLID: A system for exact boundary evaluation. *Comp. Aided Design*, 36(2):175–193, 2004.
16. D. Lazard. Quantifier elimination: optimal solution for two classical examples. *J. Symb. Comput.*, 5(1-2):261–266, 1988.
17. R. Rioboo. Real algebraic closure of an ordered field: implementation in axiom. In *Proc. Annual ACM ISSAC*, pages 206–215. ACM Press, 1992.
18. F. Rouillier and P. Zimmermann. Efficient isolation of a polynomial real roots. Technical Report 4113, INRIA–Lorraine, 2001.
19. G. Salmon. *Lessons Introductory to the Modern Higher Algebra*. Chelsea Publishing Company, New York, 1885.
20. S. Schmitt. The diamond operator for real algebraic numbers. Technical Report ECG-TR-243107-01, MPI Saarbrücken, 2003.
21. T. W. Sederberg and G.-Z. Chang. Isolating the real roots of polynomials using isolator polynomials. In C. Bajaj, editor, *Algebraic Geometry and Applications*. Springer, 1993.
22. V. Weispfenning. Quantifier elimination for real algebra—the cubic case. In *Proc. Annual ACM ISSAC*, pages 258–263. ACM Press, 1994.
23. L. Yang. Recent advances on determining the number of real roots of parametric polynomials. *J. Symbolic Computation*, 28:225–242, 1999.
24. C. Yap. *Fundamental Problems of Algorithmic Algebra*. Oxford Univ. Press, 2000.



# Code Flexibility and Program Efficiency by Genericity: Improving CGAL's Arrangements<sup>\*</sup>

Efi Fogel, Ron Wein, and Dan Halperin

School of Computer Science  
Tel Aviv University  
{efif,wein,danha}@post.tau.ac.il

**Abstract.** Arrangements of planar curves are fundamental structures in computational geometry. We describe the recent developments in the arrangement package of CGAL, the Computational Geometry Algorithms Library, making it easier to use, to extend and to adapt to a variety of applications. This improved flexibility of the code does not come at the expense of efficiency as we mainly use generic-programming techniques, which make dexterous use of the compilation process. To the contrary, we expedited key operations as we demonstrate by experiments.

## 1 Introduction

Given a set  $\mathcal{C}$  of planar curves, the *arrangement*  $\mathcal{A}(\mathcal{C})$  is the subdivision of the plane induced by the curves in  $\mathcal{C}$  into maximally connected cells. The cells can be 0-dimensional (*vertices*), 1-dimensional (*edges*) or 2-dimensional (*faces*). The *planar map* of  $\mathcal{A}(\mathcal{C})$  is the embedding of the arrangement as a planar graph, such that each arrangement vertex corresponds to a planar point, and each edge corresponds to a planar subcurve of one of the curves in  $\mathcal{C}$ . Arrangements and planar maps are ubiquitous in computational geometry, and have numerous applications (see, e.g., [8,15] for some examples), so many potential users in academia and in the industry may benefit from a generic implementation of a software package that constructs and maintains planar arrangements.

CGAL [1], the Computational Geometry Algorithms Library, is a software library, which is the product of a collaborative effort of several sites in Europe and Israel, aiming to provide a generic and robust, yet efficient, implementation of widely used geometric data structures and algorithms. The library consists of a geometric *kernel* [11,18], that in turn consists of constant-size non-modifiable geometric primitive objects (such as points, line segments, triangles etc.) and

---

<sup>\*</sup> This work has been supported in part by the IST Programs of the EU as Shared-cost RTD (FET Open) Projects under Contract No IST-2000-26473 (ECG — Effective Computational Geometry for Curves and Surfaces) and No IST-2001-39250 (MOVIE — Motion Planning in Virtual Environments), by The Israel Science Foundation founded by the Israel Academy of Sciences and Humanities (Center for Geometric Computing and its Applications), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.

predicates and operations on these objects. On top of the kernel layer, the library consists of a collection of modules, which provide implementations of many fundamental geometric data structures and algorithms. The arrangement package is a part of this layer.

In the classic computational geometry literature two assumptions are usually made to simplify the design and analysis of geometric algorithms: First, inputs are in “general position”. That is, there are no degenerate cases (e.g., three curves intersecting at a common point) in the input. Secondly, operations on real numbers yield accurate results (the “real RAM” model, which also assumes that each basic operation takes constant time). Unfortunately, these assumptions do not hold in practice. Thus, an algorithm implemented from a textbook may yield incorrect results, get into an infinite loop or just crash while running on a degenerate, or nearly degenerate, input (see [25] for examples).

The need for robust software implementation of computational-geometry algorithms has driven many researches to develop variants of the classic algorithms that are less susceptible to degenerate inputs over the last decade. At the same time, advances in computer algebra enabled the development of efficient software libraries that offer exact arithmetic manipulations on unbounded integers, rational numbers (GMP — Gnu’s multi-precision library [5]) and even algebraic numbers (the CORE [2] library and the numerical facilities of LEDA [6]). These exact *number types* serve as fundamental building-blocks in the robust implementation of many geometric algorithms.

Keyser et al. [21] implemented an arrangement-construction module for algebraic curves as part of the MAPC library. However, their implementation makes some general position assumptions. The LEDA library [6] includes geometric facilities that allow the construction and maintenance of planar maps of line segments. LEDA-based implementations of arrangements of conic curves and of cubic curves were developed under the EXACUS project [3].

CGAL’s arrangement package was the first generic software implementation, designed for constructing arrangements of arbitrary planar curves and supporting operations and queries on such arrangements. More details on the design and implementation of this package can be found in [12,17]. In this paper we summarize the recent efforts that have been put into the arrangement package and show the improvements achieved: A software design relying on the generic-programming paradigm that is more modular and easy to use, and an implementation which is more extensible, adaptable, and efficient.

The rest of this paper is organized as follows: Section 2 provides the required background on CGAL’s arrangement package introducing its architecture key-points, with a special attention to the *traits* concept. In Section 3 we demonstrate the use of generic programming techniques to improve modularity and flexibility or gain functionality that did not exist in the first place. In Section 4 we describe the principle of a meta-traits class and show how it considerably simplifies the curve hierarchy of the arrangement (as introduced in [17]). We present some experimental results in Section 5. Finally, concluding remarks and future-research suggestions are given in Section 6.

## 2 Preliminaries

### 2.1 The Arrangement Module Architecture

The `Planar_map_2<Dcel, Traits>`<sup>1</sup> class-template represents the planar embedding of a set of  $x$ -monotone planar curves that are pairwise disjoint in their interiors. It is derived from the `Topological_map` class, which provides the necessary combinatorial capabilities for maintaining the planar graph, while associating geometric data with the vertices, edges and faces of the graph. The planar map is represented using a *doubly-connected edge list* (DCEL for short), a data structure that enables efficient maintenance of two-dimensional subdivisions (see [8,20]).

The `Planar_map_2` class-template should be instantiated with two parameters. A DCEL class, which represents the underlying topological data structure, and a *traits* class, which provides the geometric functionality, and is tailored to handle a specific family of curves. It encapsulates implementation details, such as the number type used, the coordinate representation and the geometric or algebraic computation methods. The two template parameters enable the separation between the topological and geometric aspects of the planar subdivision. This separation is advantageous as it allows users to employ the package with their own special type of curves, without having any expertise in computational geometry. They should only be capable of supplying the traits methods, which mainly involve algebraic computations. Indeed, several of the package users are not familiar with computational-geometry techniques and algorithms.

The `Planar_map_with_intersections_2` class-template, should be instantiated with a `Planar_map_2` class. It inherits from the planar-map class and extends its functionality by enabling insertion of intersecting and not necessarily  $x$ -monotone curves. The main idea is to break each input curve into several  $x$ -monotone subcurves, then treat each subcurve separately. Each subcurve is in turn split at its intersection points with other curves. The result is a set of  $x$ -monotone and pairwise disjoint subcurves that induce a planar subdivision, equivalent to the arrangement of the original input curves [12]. An arrangement of a set of curves can be constructed *incrementally*, inserting the curves one by one, or *aggregately*, using a sweep-line algorithm (see, e.g., [8]). Once the arrangement is constructed, *point-location* queries on it can be answered, using different point-location strategies (see [12] for more details). Additional interface functions that modify, traverse, and display the map are available as well.

The `Arrangement_2` class-template allows the construction of a planar map with intersections, while maintaining a hierarchy of curve history. At the top level of the hierarchy stand the input curves. Each curve is subdivided into several  $x$ -monotone subcurves, forming the second level of the hierarchy. As indicated above, these  $x$ -monotone subcurves are further split such that there is no pair of subcurves that intersect in their interior. These subcurves comprise the low level of the curve hierarchy. See [17] for more details regarding the design of the `Arrangement_2` template. The curve hierarchy is essential in a variety of applications that use arrangements. Robot motion planning is one example.

<sup>1</sup> CGAL prescribes the suffix `_2` for all data structures of planar objects as a convention.

## 2.2 The Traits Class

As mentioned in the previous subsection, the `Planar_map_2` class template is parameterized with a *traits* class that defines the abstract interface between planar maps and the geometric primitives they use. The name “traits” was given by Myers [23] for a concept of a class that should support certain predefined methods, passed as a parameter to another class template. In our case, the geometric traits-class defines the family of curves handled. Moreover, details such as the number type used to represent coordinate values, the type of coordinate system used (Cartesian, homogeneous, polar), the algebraic methods used and whether extraneous data is stored with the geometric objects, are all determined by the traits. A class that follows the geometric traits-class concept defines two types of objects, namely `X_monotone_curve_2` and `Point_2`. The former represents an  $x$ -monotone curve, and the latter is the type of the endpoints of the curves, representing a point in the plane. In addition, the concept lists a minimal set of predicates on objects of these two types, sufficient to enable the operations provided by the `Planar_map_2` class:

1. Compare two points by their  $x$ -coordinates only or lexicographically (by their  $x$  and then by their  $y$ -coordinates).
2. Given an  $x$ -monotone curve  $C$  and a point  $p = (x_0, y_0)$  such that  $x_0$  is in the  $x$ -range of  $C$  (namely  $x_0$  lies between the  $x$ -coordinates of  $C$ 's endpoints), determine if  $p$  is above, below or lies on  $C$ .
3. Compare the  $y$ -values of two  $x$ -monotone curves  $C_1, C_2$  at a given  $x$ -value in the  $x$ -range of both curves.
4. Given two  $x$ -monotone curves  $C_1, C_2$  and one of their intersection points  $p$ , determine the relative positions of the two curves immediately to the right of  $p$ , or immediately to the left of  $p$ .<sup>2</sup>

In order to support the construction of an arrangement of curves (more precisely, a `Planar_map_with_intersections_2`) one should work with a refined traits class. In addition to the requirements of the planar map traits concept, it defines a third type that represents a general (not necessarily  $x$ -monotone) curve in the plane, named `Curve_2`. An intersection point of the curves is of type `Point_2`. It also lists a few more predicates and geometric constructions on the three types as follows:

1. Given a curve  $C$ , subdivide it into simple  $x$ -monotone subcurves  $C_1, \dots, C_k$ .
2. Given two  $x$ -monotone curves  $C_1, C_2$  and a point  $p$ , find the next intersection point of the two curves that is lexicographically larger, or lexicographically smaller, than  $p$ . In degenerate situations, determine the overlap between the two curves.

We include several traits classes with the public distribution of CGAL: A traits class for line segments; a traits class that operates on polylines, namely

---

<sup>2</sup> Notice that when we deal with curved objects the intersection point may also be a tangency point, so the relative position of the curves to the right of  $p$  may be the same as it was to its left.

continuous piecewise linear curves [16]; and a traits class that handles conic arcs — segments of planar algebraic curves of degree 2 such as ellipses, hyperbolas or parabolas [26]. There are other traits classes that were developed in other sites [13] and are not part of the public distribution. Many users (see, e.g., [7, 10, 14, 19, 24]) have employed the arrangement package to develop a variety of applications.

### 3 Genericity — The Name of the Game

In this section we describe how we exploited several generic-programming techniques to make our arrangement package more modular, extensible, adaptable, and efficient. We have tightened the requirements from the traits concept, and allowed for an alternative subset of requirements to be fulfilled using a tag-dispatching mechanism, enabling easier development of external traits classes. At the same time, we have also improved the performance of the built-in traits classes and extended their usability through deeper template nesting.

#### 3.1 Flexibility by Genericity

When constructing an arrangement using the sweep-line algorithm, we sweep the input set of planar curves from left to right, so it is sufficient to find just the next intersection point of a pair of curves to the right of a given reference point, and to compare two curves to the right (and not to the left) of their intersection point.<sup>3</sup> It is therefore sufficient for our arrangement traits-class to provide just a subset of the requirements listed in Section 2.2. However, even if one uses the incremental construction algorithm, one may wish to implement a reduced set of traits class methods in order to avoid code duplication.

We use a *tag-dispatching* mechanism (see [4] for more details) to select the appropriate implementation that enables users to implement their traits class with an alternative or even reduced set of member functions. The traits class is in fact injected as a template parameter into a traits-class *wrapper*, which also inherits from it. The wrapper serves as a mediator between the planar-map general operations and the traits-class primitive operations. It uses the basic set of methods provided by the base traits-class to implement a wider set of methods, using tags to identify the missing or rather the existing basic methods.

The tag `Has_left_category`, for example, indicates whether the requirements for the two methods below are satisfied:

1. Given two  $x$ -monotone curves  $C_1$ ,  $C_2$  and one of their intersection points  $p$ , determine the relative positions of the two curves immediately to the left of  $p$ .

---

<sup>3</sup> At first glance, it may seem that implementing the next intersection to the left computation is a negligible effort once we implement the next intersection to the right computation. However, for some sophisticated traits classes, such as the one described in [9], it is a major endeavor.

2. Given two  $x$ -monotone curves  $C_1, C_2$  and a point  $p$ , find the next intersection point of the two curves that is lexicographically smaller than  $p$ .

The tag `Has_reflect_category` indicates whether an alternative requirement is satisfied. That is, whether functions that reflect a point or a curve about the major axes are provided. When the *has-left* tag is false and the *reflect* tag is true, the next intersection point, or a comparison to the left of a reference point, is computed with the aid of the alternative methods by reflecting the relevant objects, performing the desired operation to the right, and then reflecting the results back. This way the user is exempt from providing an implementation of the “left” methods. If none of these methods are provided, we resort to (somewhat less efficient) algorithms based just on the reduced set of provided methods: We locate a new reference point to the left of the original point, and check what happens to its right.

### 3.2 Efficiency by Genericity

In geometric computing there is a major difference between algorithms that evaluate predicates only, and algorithms that in addition construct new geometric objects. A predicate typically computes the sign of an expression used by the program control, while a constructor produces a new geometric object such as the intersection point of two segments. If we use an exact number type to ensure robustness, the newly constructed objects often have a more complex representation in comparison with the input objects (i.e. the bit-length needed for their representation is often larger). Unless the overall algorithm is carefully designed to deal with these new objects, constructions will have a severe impact on the algorithm performance.

The `Arr_segment_traits_2` class is templated with a geometric *kernel* object, that conforms to the CGAL kernel-concept [18], and supplies all the data types, predicates and constructions on linear objects. A natural candidate is CGAL’s Cartesian kernel, which represents each segment using its two endpoints, while each point is represented using two rational coordinates. This simple representation is very natural, yet it may lead to a cascaded representation of intersection points with exponentially long bit-length (see Figure 1 for an illustration).<sup>4</sup>

To avoid this cascading problem, we introduced the `Arr_cached_segment_traits_2` class. This traits class is also templated by a geometric kernel, but uses its own internal representation of a segment: In addition to the two endpoints it also stores the coefficients of the underlying line. When a segment is split, the underlying line of the two resulting sub-segments remains the same and only their endpoints are updated. When we compute an intersection point of two segments, we use the coefficients of the corresponding underlying lines and thus overcome the undesired effect of cascading.

---

<sup>4</sup> A straightforward solution would be to normalize all computations. However, our experience shows that indiscriminate normalization considerably slows down the arrangement construction.

The *cached* segment traits-class achieves faster running times than the kernel segment traits, when arrangements with relatively many intersection points are constructed. It also allows for working with less accurate, yet computationally efficient number types, such as `Quotient<MP_Float>`<sup>5</sup> (see CGAL’s manual at [1]). On the other hand, it uses more space and stores extra data with each segment, so constructing sparse arrangements could be more efficient with the kernel (non-cached) traits-class implementation. As our software is generic, users can easily switch between the two traits classes and check which one is more suitable for their application by changing just a few lines of code. An experimental comparison of the two types of segment traits-classes is presented in Section 5.

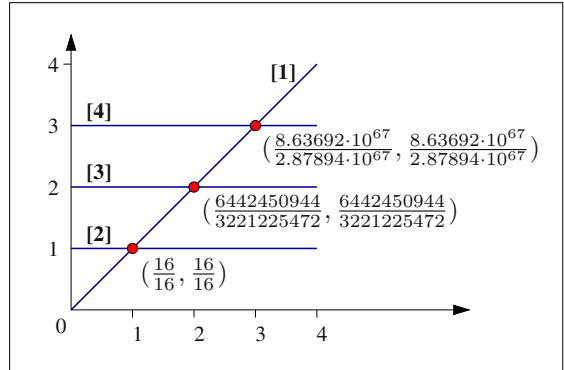
### 3.3 Succinctness by Genericity

Polylines are of particular interest, as they can be used to approximate higher-degree algebraic curves, and at the same time they are easier to deal with in comparison with conics for example.<sup>6</sup>

Previous releases of CGAL included a stand-alone polyline traits class. This class represented a polyline as a list of points and performed all geometric operations on this list (see [16] for more details). We have recently rewritten the polyline traits-class, making it a class template named `Arr_polyline_traits_2`,

that is parametrized with a `Segment_traits`, a traits

class appropriate for handling segments. The polyline is implemented as a vector of `Segment_traits::Curve_2` objects (namely of segments). The new polyline traits-class does not perform any geometric operation directly. Instead, it relies solely on the functionality of the segment traits. For example, when we wish to determine the position of a point with respect to an  $x$ -monotone polyline, we use binary search to locate the relevant segment that contains the point in its  $x$ -range, then we compare the point to this segment. Operations on polylines of size  $m$  therefore take  $O(\log m)$  time.



**Fig. 1.** Cascaded computation of the coordinates of the intersection points in an arrangement of four segments. The order of insertion of the segments is indicated in brackets. Notice the exponential growth of the bitlengths of the intersection-point coordinates.

<sup>5</sup> `MP_Float` represents floating-point numbers with an unbounded mantissa, but with a bounded exponent. In some cases (see, e.g. Figure 1) the exponent may overflow.

<sup>6</sup> With polylines it is sufficient to use an exact rational number type.

Users are free to choose the underlying segment traits, giving them the ability to use the kernel (non-cached) segment traits or the cached segment traits, depending on the number of expected intersection points. Moreover, it is possible to instantiate the polyline traits template with a *data traits-class* that handles segments with some additional data (see the next section). This makes it possible to associate some data with the entire polyline and possibly different data with each of the segments of the set that comprises it.

## 4 The Data Meta-traits

Additional information can be maintained either by extending the vertex, half-edge, or face types provided by the topological map through inheritance, or alternatively by extending their geometric mappings — that is, the point and curve types. The former option should be used to retain additional information related to the planar-map topology, and is done by instantiating an appropriate DCEL, which can be conveniently derived from the one provided with the CGAL distribution. The latter option can be carried out by extending the geometric types of the kernel, as the kernel is fully adaptable and extensible [18], but this indiscriminating extension could lead to an undue space consumption. Extending the curve type of the planar-map only is easy with the meta traits which we describe next.

We define a simple yet powerful meta-traits class template called *data traits*. The data traits-class is used to extend planar-map curve types with additional data. It should be instantiated with a regular traits-class, referred to as the *base traits-class*, and a class that contains all extraneous data associated with a curve.

```
template <class Base_traits, class Data>
class Arr_curve_data_traits_2 : public Base_traits {
public:
    // (1) Type definitions.
    // (2) Overridden functions.
};
```

The base traits-class must meet all the requirements detailed in Section 2.2 including the definition of the types `Point_2`, `Curve_2`, and `X_monotone_curve_2`. The data traits-class redefines its `Curve_2` and `X_monotone_curve_2` types as derived classes from the respective types in the base traits-class, with an additional *data* field.

```
// (1) Type definitions:
typedef Base_traits::Curve_2          Base_curve_2;
typedef Base_traits::X_monotone_curve_2 Base_x_mon_curve_2;
typedef Base_traits::Point_2          Point_2;

class Curve_2 : public Base_curve_2 {
    Data m_data;          // Additional data.
public:
```



```

Curve_2 (const Base_curve_2& cv, const Data& dat);
const Data& get_data () const;
void set_data (const Data& data);
};

class X_monotone_curve_2 : public Base_x_mon_curve_2 {
    Data m_data;          // Additional data.
public:
    X_monotone_curve_2 (const Base_x_mon_curve_2& cv, const Data& dat);
    const Data& get_data () const;
    void set_data (const Data& data);
};

```

The base traits-class must support all necessary predicates and geometric constructions on curves of the specific family it handles. The data traits-class inherits from the base traits-class all the geometric predicates and some of the geometric constructions of point objects. It only has to override the two functions that deal with constructions of  $x$ -monotone curves:

- It uses the `Base_traits::curve_make_x_monotone()` function to subdivide the basic curve into basic  $x$ -monotone subcurves. It then constructs the output subcurves by copying the data from the original curve to each of the output  $x$ -monotone subcurves.
- Similarly, the `Base_traits::curve_split()` function is used to split an  $x$ -monotone curve, then its data is copied to each of the two resulting subcurves.

```

// (2) Overridden functions:
template <class Output_iterator>
void curve_make_x_monotone (const Curve_2& cv,
                           Output_iterator& x_cvs) const;

void curve_split (const X_monotone_curve_2& cv, const Point_2& p,
                 X_monotone_curve_2& c1, X_monotone_curve_2& c2) const;

```

## 4.1 Constructing the Arrangement Hierarchy

Using the data traits-class with the appropriate parameters, it is simple to implement the arrangement hierarchy (see Section 2.1) without any additional data structures. Given a set of input curves, we construct a planar map that represents their planar arrangement. Since we want to be able to identify the originating input curve of each half-edge in the map, each subcurve we create is extended with a pointer to the input curve it originated from, using the data-traits mechanism as follows: Suppose that we have a base traits-class that supplies all the geometric methods needed to construct a planar arrangement of curves of some specific kind (e.g., segments or conic arcs). We define a data traits-class in the following form:

```

Arr_curve_data_traits_2<Base_traits, Base_traits::Curve_2 *> traits;

```

When constructing the arrangement, we keep all our base input-curves in a container. Each curve we insert to the arrangement is then formed of a base curve and a pointer to this base curve. Each time a subcurve is created, the pointer to the base input-curve is copied, and can be easily retrieved later.

## 4.2 An Additional Example

Suppose that we are given a few sets of data for some country: A geographical map of the country divided into regions, the national road and railroad network, and the water routes. Roads, railroads, and rivers are represented as polylines and have attributes (e.g., a name). We wish to obtain all crossroads and all bridges in some region of this country.

Using the data traits we can give a straightforward solution to this problem. Suppose that the class `Polyline_traits_2` supplies all the necessary geometric type definition and methods for polylines, fulfilling the requirements of the traits concept. We define the following classes:

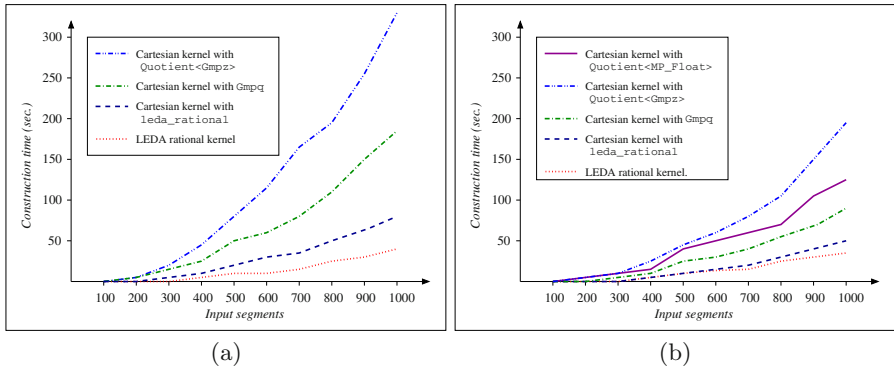
```
struct My_data {
    enum {ROAD, RAILROAD, RIVER} type;
    std::string      name;
};
Arr_curve_data_traits_2<Polyline_traits_2, My_data> traits;
```

Each curve consists of a base polyline-curve (e.g., a road, a river) and a name. We construct the arrangement of all curves in our datasets overlayed on top of the regional map. Then, we can simply go over all arrangement vertices located in the desired region, and examine the half-edges around each vertex. If we find only `ROAD` or `RAILROAD` half-edges around a vertex, we can conclude it represents a crossroad. A vertex where half-edges of types `ROAD` (or `RAILROAD`) and `RIVER` meet represents a bridge. In any case, we can easily retrieve the names of the intersecting roads or rivers and present them as part of the output.

## 5 Experimental Results

As mentioned above, there are many ways to represent line segments in the plane. In the first set of experiments we compared the performance of some representations. When the CGAL Cartesian kernel is used as the template parameter of the traits class (`Arr_segment_traits_2` for example), the user is still free to choose among different number types. We conducted our experiments with (i) `Quotient<MP_Float>`, (ii) `Quotient<Gmpz>`, representing the numerator and denominator as two unbounded integers, (iii) `Gmpq`, GMP's rational class, and (iv) `Leda_rational`, an efficient implementation of exact rationals. In addition, we used an external geometric kernel, called LEDA rational kernel [22], that uses floating-point filtering to speed up computations.

We have tested each representation on ten input sets, containing 100–1000 random input segments, having a quadratic number of intersection points. The results are summarized in Figure 2. The cached traits-class achieves better



**Fig. 2.** Construction times for arrangements of random line segments: (a) Using the kernel (non-cached) segment traits, (b) using the cached segment traits.

performance for all tested configurations. Moreover, it is not possible to construct an arrangement using the `Quotient<MP_Float>` number type with the kernel traits (see Section 3.2). For lack of space, we do not show experiments with sparse arrangements.

It is worth mentioning that switching from one configuration to another requires a change of just a few lines of code. In fact, we used a benchmarking toolkit that automatically generates all the required configurations and measures the performance of each configuration on a set of inputs.

Figure 3 shows the output of the algorithm presented in Section 4.2. The input set, consisting of more than 900 polylines, represents major roads, railroads, rivers and water-canal in the Netherlands. The arrangement construction takes just a few milliseconds.



**Fig. 3.** Roads, railroads, rivers and water canals on the map of the Netherlands. Bridges are marked by circles.

## 6 Conclusions and Future Work

We show how our arrangement package can be used with various components and different underlying algorithms that can be plugged in using the appropriate traits class. Users may select the configuration that is most suitable for their application from the variety offered in CGAL or in its accompanying software

libraries, or implement their own traits class. Switching between different traits classes typically involves just a minor change of a few lines of code.

We have shown how careful software design based on the generic-programming paradigm makes it easier to adapt existing traits classes or even to develop new ones. We believe that similar techniques can be employed in other software packages from other disciplines as well.

In the future we plan to augment the polyline traits-class into a traits class that handles piecewise general curves that are not necessarily linear, and provide means to extend the planar-map point geometric type in similar ways the curve data-traits extends the curve type.

## References

1. The CGAL project homepage. <http://www.cgal.org/>.
2. The CORE library homepage. <http://www.cs.nyu.edu/exact/core/>.
3. The EXACUS homepage. <http://www.mpi-sb.mpg.de/projects/EXACUS/>.
4. Generic programming techniques.  
[http://www.boost.org/more/generic\\_programming.html](http://www.boost.org/more/generic_programming.html).
5. The GNU MP bignum library. <http://www.swox.com/gmp/>.
6. The LEDA homepage. <http://www.algorithmic-solutions.com/enleda.htm>.
7. D. Cohen-Or, S. Lev-Yehudi, A. Karol, and A. Tal. Inner-cover of non-convex shapes. *International Journal on Shape Modeling*, 9(2):223–238, Dec 2003.
8. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.
9. O. Devillers, A. Fronville, B. Mourrain, and M. Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. *Comput. Geom. Theory Appl.*, 22(1–3):119–142, 2002.
10. D. A. Duc, N. D. Ha, and L. T. Hang. Proposing a model to store and a method to edit spatial data in topological maps. Technical report, Ho Chi Minh University of Natural Sciences, Ho Chi Minh City, Vietnam, 2001.
11. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the Computational Geometry Algorithms Library. *Software — Practice and Experience*, 30:1167–1202, 2000.
12. E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The design and implementation of planar maps in CGAL. *The ACM Journal of Experimental Algorithmics*, 5, 2000. Also in LNCS Vol. 1668 (WAE '99), pages 154–168.
13. E. Fogel et al. An empirical comparison of software for constructing arrangements of curved arcs. Technical Report ECG-TR-361200-01, Tel-Aviv Univ., 2004.
14. B. Gerkey. Visibility-based pursuit-evasion for searchers with limited field of view. Presented in the 2nd CGAL User Workshop (2004).
15. D. Halperin. Arrangements. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. Chapman & Hall/CRC, 2nd edition, 2004.
16. I. Hanniel. The design and implementation of planar arrangements of curves in CGAL. M.Sc. thesis, School of Computer Science, Tel Aviv University, 2000.
17. I. Hanniel and D. Halperin. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *LNCS Vol. 1982 (Proc. WAE '00)*, pages 171–182. Springer-Verlag, 2000.

18. S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *LNCS Vol. 2141 (Proc. WAE '01)*, pages 79–90. Springer-Verlag, 2001.
19. S. Hirsch and D. Halperin. Hybrid motion planning: Coordinating two discs moving among polygonal obstacles in the plane. In J.-D. Boissonnat, J. Burdick, K. Goldberg, and S. Hutchinson, editors, *Algorithmic Foundations of Robotics V*, pages 239–255. Springer, 2003.
20. L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theory Appl.*, 13:65–90, 1999.
21. J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: a library for efficient manipulation of algebraic points and curves. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 360–369, 1999. <http://www.cs.unc.edu/~geom/MAPC/>.
22. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.
23. N. Myers. Traits: A new and useful template technique. *C++ Gems*, 17, 1995.
24. V. Rogol. Maximizing the area of an axially-symmetric polygon inscribed by a simple polygon. Master's thesis, Technion, Haifa, Israel, 2003.
25. S. Schirra. Robustness and precision issues in geometric computation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1999.
26. R. Wein. High-level filtering for arrangements of conic arcs. In *Proc. ESA 2002*, pages 884–895. Springer-Verlag, 2002.

# Finding Dominators in Practice<sup>\*</sup>

Loukas Georgiadis<sup>1</sup>, Renato F. Werneck<sup>1</sup>, Robert E. Tarjan<sup>1,2</sup>,  
Spyridon Triantafyllis<sup>1</sup>, and David I. August<sup>1</sup>

<sup>1</sup> Dept. of Computer Science, Princeton University, Princeton NJ, 08544, USA  
<sup>2</sup> Hewlett-Packard, Palo Alto, CA

**Abstract.** The computation of dominators in a flowgraph has applications in program optimization, circuit testing, and other areas. Lengauer and Tarjan [17] proposed two versions of a fast algorithm for finding dominators and compared them experimentally with an iterative bit vector algorithm. They concluded that both versions of their algorithm were much faster than the bit-vector algorithm even on graphs of moderate size. Recently Cooper et al. [9] have proposed a new, simple, tree-based iterative algorithm. Their experiments suggested that it was faster than the simple version of the Lengauer-Tarjan algorithm on graphs representing computer program control flow. Motivated by the work of Cooper et al., we present an experimental study comparing their algorithm (and some variants) with careful implementations of both versions of the Lengauer-Tarjan algorithm and with a new hybrid algorithm. Our results suggest that, although the performance of all the algorithms is similar, the most consistently fast are the simple Lengauer-Tarjan algorithm and the hybrid algorithm, and their advantage increases as the graph gets bigger or more complicated.

## 1 Introduction

A flowgraph  $G = (V, A, r)$  is a directed graph with  $|V| = n$  vertices and  $|A| = m$  arcs such that every vertex is reachable from a distinguished root vertex  $r \in V$ . A vertex  $w$  *dominates* a vertex  $v$  if every path from  $r$  to  $v$  includes  $w$ . Our goal is to find for each vertex  $v$  in  $V$  the set  $Dom(v)$  of all vertices that dominate  $v$ . Certain applications require computing the *postdominators* of  $G$ , defined as the dominators in the graph obtained from  $G$  by reversing all arc orientations.

Compilers use dominance information extensively during program analysis and optimization, for such diverse goals as natural loop detection (which enables a host of optimizations), structural analysis [20], scheduling [22], and the computation of dependence graphs and static single-assignment forms [10]. Dominators are also used to identify pairs of equivalent line faults in VLSI circuits [7].

The problem of finding dominators has been extensively studied. In 1972 Allen and Cocke showed that the dominance relation can be computed iteratively from a set of data-flow equations [5]. A direct implementation of this solution

---

<sup>\*</sup> L. Georgiadis, R.F. Werneck and R.E. Tarjan partially supported by the Aladdin project, NSF Grant No. CCR-9626862.

has  $O(mn^2)$  worst-case time. Purdom and Moore [18] gave a straightforward algorithm with complexity  $O(mn)$ . It consists of performing a search in  $G - v$  for all  $v \in V$  ( $v$  obviously dominates all the vertices that become unreachable). Improving on previous work by Tarjan [23], Lengauer and Tarjan [17] proposed an  $O(m\alpha(m, n))$ -time algorithm, where  $\alpha(m, n)$  is an extremely slow-growing functional inverse of the Ackermann function. Alstrup et al. [6] gave a linear-time solution for the random-access model; a simpler solution was given by Buchsbaum et al. [8]. Georgiadis and Tarjan [12] achieved the first linear-time algorithm for the pointer-machine model.

Experimental results for the dominators problem appear in [17,8,9]. In [17] Lengauer and Tarjan found the almost-linear-time version of their algorithm (LT) to be faster than the simple  $O(m \log n)$  version even for small graphs. They also show that Purdom-Moore [18] is only competitive for graphs with fewer than 20 vertices, and that a bit-vector implementation of the iterative algorithm, by Aho and Ullman [4], is 2.5 times slower than LT for graphs with more than 100 vertices. Buchsbaum et al. [8] show that their claimed linear-time algorithm has low constants, being only about 10% to 20% slower than LT for graphs with more than 300 vertices. This algorithm was later shown to have the same time complexity as LT [12], and the corrected version is more complicated (see Corrigendum of [8]). Cooper et al. [9] present a space-efficient implementation of the iterative algorithm, which they claimed to be 2.5 times faster than the simple version of LT. However, a more careful implementation of LT later led to different results (personal communication).

In this paper we cast the iterative algorithm into a more general framework and explore the effects of different initializations and processing orderings. We also discuss implementation issues that make both versions of LT faster in practice and competitive with simpler algorithms even for small graphs. Furthermore, we describe a new algorithm that combines LT with the iterative algorithm and is very fast in practice. Finally, we present a thorough experimental analysis of various algorithms using real as well as artificial data. We did not include linear-time algorithms in our study; they are significantly more complex and thus unlikely to be faster than LT in practice.

## 2 Algorithms

The *immediate dominator* of a vertex  $v$ , denoted by  $\text{idom}(v)$ , is the unique vertex  $w \neq v$  that dominates  $v$  and is dominated by all vertices in  $\text{Dom}(v) - v$ . The (immediate) *dominator tree* is a directed tree  $I$  rooted at  $r$  and formed by the edges  $\{(\text{idom}(v), v) \mid v \in V - r\}$ . A vertex  $w$  dominates  $v$  if and only if  $w$  is a proper ancestor of  $v$  in  $I$  [3], so it suffices to compute the immediate dominators. Throughout this paper the notation “ $v \xrightarrow{*}_F u$ ” means that  $v$  is an ancestor of  $u$  in the forest  $F$  and “ $v \xrightarrow{+}_F u$ ” means that  $v$  is a proper ancestor of  $u$  in  $F$ . We omit the subscript when the context is clear. Also, we denote by  $\text{pred}(v)$  the set of predecessors in  $G$  of vertex  $v$ . Finally, for any subset  $U \subseteq V$  and a tree  $T$ ,  $\text{NCA}(T, U)$  denotes the nearest common ancestor of  $U \cap T$  in  $T$ .

## 2.1 The Iterative Algorithm

Clearly  $Dom(r) = \{r\}$ . For each of the remaining vertices, the set of dominators is the solution to the following data-flow equations:

$$Dom'(v) = \left( \bigcap_{u \in pred(v)} Dom'(u) \right) \cup \{v\}, \forall v \in V - r. \quad (1)$$

Allen and Cocke [5] showed that one can iteratively find the maximal fixed-point solution  $Dom'(v) = Dom(v)$  for all  $v$ . Typically the algorithm either initializes  $Dom'(v) \leftarrow V$  for all  $v \neq r$ , or excludes uninitialized  $Dom'(u)$  sets from the intersection in (1). Cooper et al. [9] observe that the iterative algorithm does not need to keep each  $Dom'$  set explicitly; it suffices to maintain the transitive reduction of the dominance relation, which is a tree  $T$ . The intersection of  $Dom'(u)$  and  $Dom'(w)$  is the path from  $NCA(T, \{u, w\})$  to  $r$ . Any spanning (sub)tree  $S$  of  $G$  rooted at  $r$  is a valid initialization for  $T$ , since for any  $v \in S$  only vertices in  $r \xrightarrow{*}_S v$  can dominate  $v$ .

It is known [19] that the dominator tree  $I$  is such that if  $w$  is a vertex in  $V - r$  then  $idom(w) = NCA(I, pred(w))$ . Thus the iterative algorithm can be interpreted as a process that modifies a tree  $T$  successively until this property holds. The number of iterations depends on the order in which the vertices (or edges) are processed. Kam and Ullman [15] show that certain dataflow equations, including (1), are solved in up to  $d(G, D) + 3$  iterations when the vertices are processed in reverse postorder with respect to a DFS tree  $D$ . Here  $d(G, D)$  is the *loop connectedness of  $G$  with respect to  $D$* , the largest number of back edges found in any cycle-free path of  $G$ . When  $G$  is reducible [13] the dominator tree is built in one iteration, because  $v$  dominates  $u$  whenever  $(u, v)$  is a back edge.

The running time per iteration is dominated by the time spent on NCA calculations. If they are performed naïvely (ascending the tree paths until they meet), then a single iteration costs  $O(mn)$  time. Because there may be up to  $O(n)$  iterations, the running time is  $O(mn^2)$ . The iterative algorithm runs much faster in practice, however. Typically  $d(G, D) \leq 3$  [16], and it is reasonable to expect that few NCA calculations will require  $O(n)$  time. If  $T$  is represented as a dynamic tree [21], the worst-case bound per iteration is reduced to  $O(m \log n)$ , but the implementation becomes much more complicated.

**Initializations and vertex orderings.** Our base implementation of the iterative algorithm (IDFS) starts with  $T \leftarrow \{r\}$  and processes the vertices in reverse postorder with respect to a DFS tree, as done in [9]. This requires a preprocessing phase that executes a DFS on the graph and assigns a postorder number to each vertex. Initializing  $T$  as a DFS tree is bad both in theory and in practice because it causes the back edges to be processed, even though they contribute nothing to the NCAs. Intuitively, a much better initial approximation of the dominator tree is a BFS tree. We implemented a variant of the iterative algorithm (which we call IBFS) that starts with such a tree and processes the vertices in BFS order. As Section 4 shows, this method is often (but not always) faster than IDFS.



Finally, we note that there is an ordering  $\sigma$  of the edges which is optimal with respect to the number of iterations that are needed for convergence. If we initialize  $T = \{r\}$  and process the edges according to  $\sigma$ , then after one iteration we will have constructed the dominator tree. We are currently investigating if such an ordering can be found efficiently.

## 2.2 The Lengauer-Tarjan Algorithm

The Lengauer-Tarjan algorithm starts with a depth-first search on  $G$  from  $r$  and assigns preorder numbers to the vertices. The resulting DFS tree  $D$  is represented by an array *parent*. For simplicity, we refer to the vertices of  $G$  by their preorder number, so  $v < u$  means that  $v$  has a lower preorder number than  $u$ . The algorithm is based on the concept of *semidominators*, which give an initial approximation to the immediate dominators. A path  $P = (u = v_0, v_1, \dots, v_{k-1}, v_k = v)$  in  $G$  is a *semidominator path* if  $v_i > v$  for  $1 \leq i \leq k-1$ . The semidominator of  $v$  is defined as  $sdom(v) = \min\{u \mid \text{there is a semidominator path from } u \text{ to } v\}$ .

Semidominators and immediate dominators are computed by finding minimum *sdom* values on paths of  $D$ . Vertices are processed in reverse preorder, which ensures that all the necessary values are available when needed. The algorithm maintains a forest  $F$  such that when it needs the minimum *sdom*( $u$ ) on a path  $p = w \xrightarrow{+} u \xrightarrow{*} v$ ,  $w$  will be the root of a tree in  $F$  containing all vertices on  $p$  (in general, the root of the tree containing  $v$  is denoted by  $root_F(v)$ ). Every vertex in  $V$  starts as a singleton in  $F$ . Two operations are defined on  $F$ : *link*( $v$ ) links the tree rooted at  $v$  to the tree rooted at *parent*[ $v$ ]; *eval*( $v$ ) returns a vertex  $u$  of minimum *sdom* among those satisfying  $root_F(v) \xrightarrow{+} u \xrightarrow{*} v$ .

Every vertex  $w$  is processed three times. First, *sdom*( $w$ ) is computed and  $w$  is inserted into a bucket associated with vertex *sdom*( $w$ ). The algorithm processes  $w$  again after *sdom*( $v$ ) has been computed, where  $v$  satisfies *parent*[ $v$ ] = *sdom*( $w$ ) and  $v \xrightarrow{*} w$ ; then it finds either the immediate dominator or a *relative dominator* of  $w$  (an ancestor of  $w$  that has the same immediate dominator as  $w$ ). Finally, immediate dominators are derived from relative dominators in a preorder pass.

With a simple implementation of *link-eval* (using path compression but not balancing), the LT algorithm runs in  $O(m \log n)$  time [25]. With a more sophisticated linking strategy that ensures that  $F$  is balanced, LT runs in  $O(m\alpha(m, n))$  time [24]. We refer to these two versions as SLT and LT, respectively.

**Implementation issues.** Buckets have very specific properties in the Lengauer-Tarjan algorithm: (1) every vertex is inserted into at most one bucket; (2) there is exactly one bucket associated with each vertex; (3) vertex  $i$  can only be inserted into some bucket after bucket  $i$  itself is processed. Properties (1) and (2) ensure that buckets can be implemented with two  $n$ -sized arrays, *first* and *next*: *first*[ $i$ ] represents the first element in bucket  $i$ , and *next*[ $v$ ] is the element that succeeds  $v$  in the bucket it belongs to. Property (3) ensures that these two arrays can be combined into a single array *bucket*.

In [17], Lengauer and Tarjan process  $\text{bucket}[\text{parent}[w]]$  at the end of the iteration that deals with  $w$ . A better alternative is to process  $\text{bucket}[w]$  in the beginning of the iteration; each bucket is now processed exactly once, so it need not be emptied explicitly. Another measure that is relevant in practice is to avoid unnecessary bucket insertions: a vertex  $w$  for which  $\text{parent}[w] = \text{sdom}(w)$  is not inserted into any bucket because we already know that  $\text{idom}(w) = \text{parent}[w]$ .

### 2.3 The SEMI-NCA Algorithm

SEMI-NCA is a new hybrid algorithm for computing dominators that works in two phases: the first computes  $\text{sdom}(v)$  for all  $v \in V - r$  (as in LT), and the second builds  $I$  incrementally, using the fact that for any vertex  $w \neq r$ ,  $\text{idom}(w) = \text{NCA}(I, \{\text{parent}[w], \text{sdom}(w)\})$  [12]. In the second phase, for every  $w$  in preorder we ascend the  $I$ -tree path from  $\text{parent}[w]$  to  $r$  until we meet the first vertex  $x$  such that  $x \leq \text{sdom}(w)$ . Then we have  $x = \text{idom}(w)$ . With this implementation, the second phase runs in  $O(n^2)$  worst-case time. However, we expect it to be much faster in practice, since our empirical results indicate that  $\text{sdom}(v)$  is usually a good approximation to  $\text{idom}(v)$ .

SEMI-NCA is simpler than LT in two ways. First, eval can return the minimum value itself rather than a vertex that achieves that value. This eliminates one array and one level of indirect addressing. Second, buckets are no longer necessary because the vertices are processed in preorder in the second phase. With the simple implementation of link-eval (which is faster in practice), this method (SNCA) runs in  $O(n^2 + m \log n)$  worst-case time. We note that Gabow [11] presents a rather complex procedure that computes NCAs in total linear time on a tree that grows by adding leaves. This implies that the second phase of SEMI-NCA can run in  $O(n)$  time, but it is unlikely to be practical.

## 3 Worst-Case Behavior

This section briefly describes families of graphs that elicit the worst-case behavior of the algorithms we implemented. Figure 1 shows graph families that favor particular methods against the others. Family  $\text{itworst}(k)$  is a graph with  $O(k)$  vertices and  $O(k^2)$  edges for which IDFS and IBFS need to spend  $O(k^4)$  time. Family  $\text{idfsquad}(k)$  has  $O(k)$  vertices and  $O(k)$  edges and can be processed in linear time by IBFS, but IDFS needs quadratic time;  $\text{ibfsquad}(k)$  achieves the reverse effect. Finally  $\text{sncaworst}(k)$  has  $O(k)$  vertices and  $O(k)$  edges and causes SNCA, IDFS and IBFS to run in quadratic time. Adding any  $(y_i, x_k)$  would make SNCA and IBFS run in  $O(k)$  time, but IDFS would still need  $O(k^2)$  time. We also define the family  $\text{sltworst}(k)$ , which causes worst-case behavior for SLT [25]. Because it has  $O(k)$  back edges, the iterative methods run in quadratic time.

## 4 Empirical Analysis

Based on worst-case bounds only, the sophisticated version of the Lengauer-Tarjan algorithm is the method of choice among those studied here. In practice,

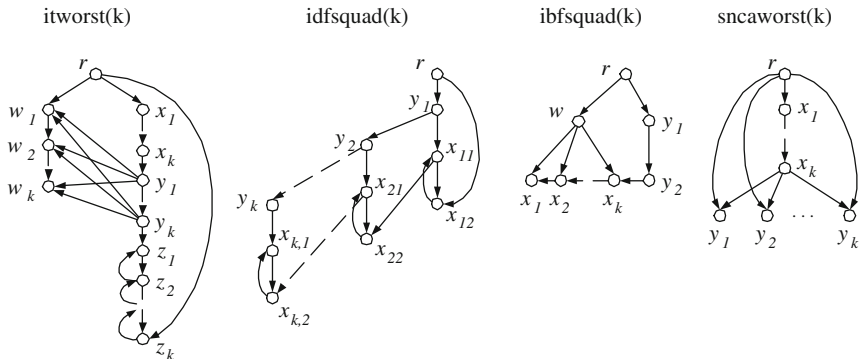


Fig. 1. Worst-case families.

however, “sophisticated” algorithms tend to be harder to code and to have higher constants, so one might prefer other alternatives. The experiments reported in this section shed some light on this issue.

**Implementation and Experimental Setup.** We implemented all algorithms in C++. They take as input the graph and its root, and return an  $n$ -element array representing immediate dominators. Vertices are assumed to be integers 1 to  $n$ . Within reason, we made all implementations as efficient and uniform as we could. The source code is available from the authors upon request.

The code was compiled using `g++ v. 3.2.2` with full optimization (flag `-O4`). All tests were conducted on a Pentium IV with 256 MB of RAM and 256 kB of cache running Mandrake Linux at 1.7 GHz. We report CPU times measured with the `getrusage` function. Since its precision is only 1/60 second, we ran each algorithm repeatedly for at least one second; individual times were obtained by dividing the total time by the number of runs. To minimize fluctuations due to external factors, we used the machine exclusively for tests, took each measurement three times, and picked the best. Running times do not include creating the graph, but they do include allocating and deallocating the arrays used by each particular algorithm.

**Instances.** We used control-flow graphs produced by the SUIF compiler [14] from benchmarks in the SPEC’95 suite [2] and previously tested by Buchsbaum et al. [8] in the context of dominator analysis. We also used control-flow graphs created by the IMPACT compiler [1] from six programs in the SPEC 2000 suite. The instances were divided into *series*, each corresponding to a single benchmark. Series were further grouped into three classes, SUIF-FP, SUIF-INT, and IMPACT. We also considered two variants of IMPACT: class IMPACTP contains the reverse graphs and is meant to test how effectively the algorithms compute postdominators; IMPACTS contains the same instances as IMPACT,

with parallel edges removed.<sup>1</sup> We also ran the algorithms on circuits from VLSI-testing applications [7] obtained from the ISCAS'89 suite [26] (all 50 graphs were considered a single class).

Finally, we tested eight instances that do not occur in any particular application related to dominators. Five are the worst-case instances described in Section 3, and the other three are large graphs representing speech recognition finite state automata (also used by Buchsbaum et al. [8]).

**Test Results.** We start with the following experiment: read an entire series into memory and compute dominators for each graph in sequence, measuring the total running time. For each series, Table 1 shows the total number of graphs ( $g$ ) and the average number of vertices and edges ( $n$  and  $m$ ). As a reference, we report the average time (in microseconds) of a simple breadth-first search (BFS) on each graph. Times for computing dominators are given as multiples of BFS.

In absolute terms, all algorithms are reasonably fast: none was more than seven times slower than BFS. Furthermore, despite their different worst-case complexities, all methods have remarkably similar behavior in practice. In no series was an algorithm twice as fast (or slow) as any other. Differences do exist, of course. LT is consistently slower than SLT, which can be explained by the complex nature of LT and the relatively small size of the instances tested. The iterative methods are faster than LT, but often slower than SLT. Both variants (IDFS and IBFS) usually have very similar behavior, although one method is occasionally much faster than the other (series 145.fppp and 256.bzip2 are good examples). Always within a factor of four of BFS, SNCA and SLT are the most consistently fast methods in the set.

By measuring the total time per series, the results are naturally biased towards large graphs. For a more complete view, we also computed running times for individual instances, and normalized them with respect to BFS. For each class, Table 2 shows the geometric mean and the geometric standard deviation of the relative times. Now that each graph is given equal weight, the aggregate measures for iterative methods (IBFS and IDFS) are somewhat better than before, particularly for IMPACT instances. Deviations, however, are higher. Together, these facts suggest that iterative methods are faster than other methods for small instances, but slower when size increases.

Figure 2 confirms this. Each point represents the mean relative running times for all graphs in the IMPACT class with the same value of  $\lceil \log_2(n + m) \rceil$ . Iterative methods clearly have a much stronger dependence on size than other algorithms. Almost as fast as a single BFS for very small instances, they become the slowest alternatives as size increases. The relative performance of the other methods is the same regardless of size: SNCA is slightly faster than SLT, and both are significantly faster than LT. A similar behavior was observed for IMPACTS and IMPACTP; for SUIF, which produces somewhat simpler graphs, iterative methods remained competitive even for larger sizes.

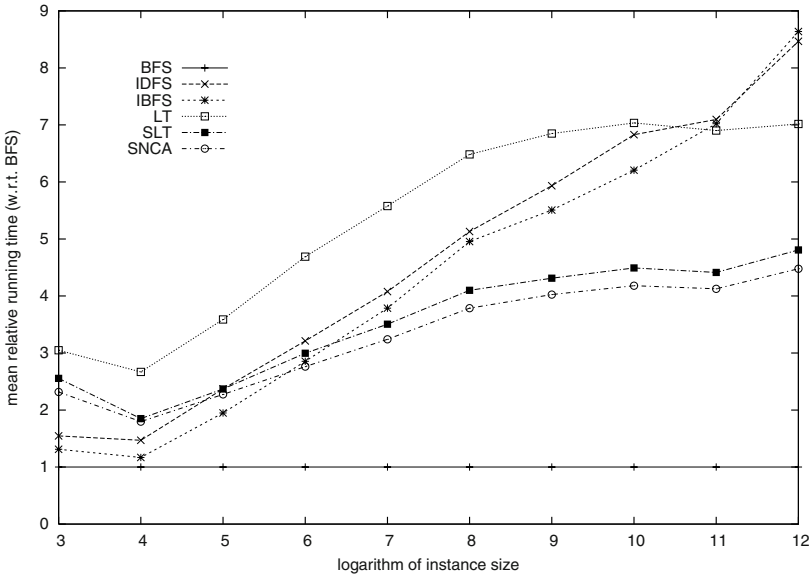
<sup>1</sup> These edges appear in optimizing compilers due to superblock formation, and are produced much more often by IMPACT than by SUIF.

**Table 1.** Complete series: number of graphs ( $g$ ), average number of vertices ( $n$ ) and edges ( $m$ ), and average time per graph (in microseconds for BFS, and relative to BFS for other methods). The best result in each row is marked in bold.

INSTANCE		DIMENSIONS			BFS	RELATIVE TOTAL TIMES				
CLASS	SERIES	$g$	$n$	$m$	TIME	IDFS	IBFS	LT	SLT	SNCA
CIRCUITS	circuits	50	3228.8	5027.2	228.88	5.41	6.35	4.98	3.80	<b>3.48</b>
IMPACT	181.mcf	26	26.5	90.3	1.41	4.75	4.36	5.20	3.33	<b>3.25</b>
	197.parser	324	16.8	55.7	1.22	4.22	3.66	4.39	3.09	<b>2.99</b>
	254.gap	854	25.3	56.2	1.88	3.12	2.88	3.87	2.71	<b>2.61</b>
	255.vortex	923	15.1	35.8	1.27	4.04	3.84	4.30	3.24	<b>3.13</b>
	256.bzip2	74	22.8	70.3	1.26	4.81	3.97	4.88	3.36	<b>3.20</b>
	300.twolf	191	39.5	115.6	2.52	4.58	4.13	5.01	3.51	<b>3.36</b>
IMPACTP	181.mcf	26	26.5	90.3	1.41	4.65	4.34	5.09	3.41	<b>3.21</b>
	197.parser	324	16.8	55.7	1.23	4.13	3.40	4.21	3.01	<b>2.94</b>
	254.gap	854	25.3	56.2	1.82	3.32	3.44	3.79	2.69	<b>2.68</b>
	255.vortex	923	15.1	35.8	1.26	4.24	4.03	4.19	<b>3.32</b>	<b>3.32</b>
	256.bzip2	74	22.8	70.3	1.28	5.03	3.73	4.78	3.23	<b>3.07</b>
	300.twolf	191	39.5	115.6	2.52	4.86	4.52	4.88	3.38	<b>3.33</b>
IMPACTS	181.mcf	26	26.5	72.4	1.30	4.36	4.04	5.22	3.30	<b>3.24</b>
	197.parser	324	16.8	42.1	1.10	4.10	3.56	4.67	3.42	<b>3.32</b>
	254.gap	854	25.3	48.8	1.75	3.02	2.82	4.00	2.80	<b>2.66</b>
	255.vortex	923	15.1	27.1	1.16	2.59	2.41	3.50	2.45	<b>2.34</b>
	256.bzip2	74	22.8	53.9	1.17	4.25	3.53	4.91	3.33	<b>3.24</b>
	300.twolf	191	39.5	96.5	2.23	4.50	4.09	5.12	3.50	<b>3.41</b>
SUIF-FP	101.tomcatv	1	143.0	192.0	4.23	<b>3.42</b>	3.90	5.78	3.67	3.66
	102.swim	7	26.6	34.4	1.04	<b>2.77</b>	3.00	4.48	2.97	2.82
	103.su2cor	37	32.3	42.7	1.29	<b>2.82</b>	2.99	4.68	3.01	3.03
	104.hydro2d	43	35.3	47.0	1.39	<b>2.79</b>	3.05	4.64	2.94	2.86
	107.mgrid	13	27.2	35.4	1.12	<b>2.58</b>	3.01	4.25	2.82	2.77
	110.applu	17	62.2	82.8	2.03	<b>3.28</b>	3.58	5.36	3.45	3.41
	125.turb3d	24	54.0	73.5	1.51	3.57	3.59	6.31	3.66	<b>3.44</b>
	145.fpppp	37	20.3	26.4	0.82	<b>3.00</b>	3.43	4.83	3.19	3.19
	146.wave5	110	37.4	50.7	1.43	<b>3.09</b>	3.11	5.00	3.22	3.15
SUIF-INT	009.go	372	36.6	52.5	1.72	3.12	3.01	4.71	<b>3.00</b>	3.07
	124.m88ksim	256	27.0	38.7	1.17	3.35	<b>3.10</b>	4.98	3.16	3.18
	126.gcc	2013	48.3	69.8	2.35	3.00	3.01	4.60	<b>2.91</b>	2.99
	129.compress	24	12.6	16.7	0.66	2.79	<b>2.46</b>	3.76	2.60	2.55
	130.li	357	9.8	12.8	0.54	2.59	<b>2.44</b>	3.92	2.67	2.68
	132.jpeg	524	14.8	20.1	0.78	2.84	<b>2.60</b>	4.35	2.84	2.82
	134.perl	215	66.3	98.2	2.74	3.77	3.76	5.43	<b>3.44</b>	3.50
	147.vortex	923	23.7	34.9	1.35	2.69	2.67	3.92	2.59	<b>2.52</b>

**Table 2.** Times relative to BFS: geometric mean and geometric standard deviation. The best result in each row is marked in bold.

CLASS	IDFS		IBFS		LT		SLT		SNCA	
	MEAN	DEV	MEAN	DEV	MEAN	DEV	MEAN	DEV	MEAN	DEV
CIRCUITS	5.89	1.19	6.17	1.42	6.71	1.18	4.62	1.15	<b>4.40</b>	1.14
SUIF-FP	2.49	1.44	<b>2.34</b>	1.58	3.74	1.42	2.54	1.36	2.96	1.38
SUIF-INT	2.45	1.50	<b>2.25</b>	1.62	3.69	1.40	2.48	1.33	2.73	1.45
IMPACT	2.60	1.65	<b>2.24</b>	1.77	4.02	1.40	2.74	1.33	2.56	1.31
IMPACTP	2.58	1.63	<b>2.25</b>	1.82	3.84	1.44	2.61	1.30	2.52	1.29
IMPACTS	2.42	1.55	<b>2.05</b>	1.68	3.62	1.33	2.50	1.28	2.61	1.45



**Fig. 2.** Times for IMPACT instances within each size. Each point represents the mean relative running time (w.r.t. BFS) for all instances with the same value of  $\lceil \log_2(n+m) \rceil$ .

The results for IMPACT and IMPACTS indicate that the iterative methods benefit the most by the absence of parallel edges. Because of path compression, Lengauer-Tarjan and SEMI-NCA can handle repeated edges in constant time.

Table 3 helps explain the relative performance of the methods with three pieces of information. The first is SDP%, the percentage of vertices (excluding the root) whose semidominators are their parents in the DFS tree. These vertices are not inserted into buckets, so large percentages are better for LT and SLT. On average, far more than half of the vertices have this property. In practice, avoiding unnecessary bucket insertions resulted in a 5% to 10% speedup.

The next two columns show the average number of iterations performed by IDFS and IBFS. It is very close to 2 for IDFS: almost always the second iteration

**Table 3.** Percentage of vertices that have their parents as semidominators (SDP%), average number of iterations and number of comparisons per vertex.

CLASS	SDP	ITERATIONS		COMPARISONS PER VERTEX					
	(%)	IDFS	IBFS	IDFS	IBFS	LT	SLT	SNCA	
CIRCUITS	76.7	2.8000	3.2000	32.6	39.3	12.0	9.9	8.9	
IMPACT	73.4	2.0686	1.4385	30.9	28.0	15.6	12.8	11.1	
IMPACTP	88.6	2.0819	1.5376	30.2	32.2	15.5	12.3	10.9	
IMPACTS	73.4	2.0686	1.4385	24.8	23.4	13.9	11.2	9.5	
SUIF-FP	67.7	2.0000	1.6817	12.3	15.9	10.3	8.3	6.8	
SUIF-INT	63.9	2.0009	1.6659	14.9	17.2	11.2	8.6	7.2	

just confirms that the candidate dominators found in the first are indeed correct. This is expected for control-flow graphs, which are usually reducible in practice. On most classes the average is smaller than 2 for IBFS, indicating that the BFS and dominator trees often coincide. Note that the number of iterations for IMPACTP is slightly higher than for IMPACT, since the reverse of a reducible graph may be irreducible.

The last five columns show how many times on average a vertex is compared to other vertices (the results do not include the initial DFS or BFS). The number of comparisons is always proportional to the total running time; what varies is the constant of proportionality, much smaller for simpler methods than for elaborate ones. Iterative methods need many more comparisons, so their competitiveness results mainly from smaller constants. For example, they need to maintain only three arrays, as opposed to six or more for the other methods. (Two of these arrays translate vertex numbers into DFS or BFS labels and vice-versa.)

**Table 4.** Individual graphs (times for BFS in microseconds, all others relative to BFS). The best result in each row is marked in bold.

NAME	INSTANCE		BFS TIME	RELATIVE RUNNING TIMES					
	VERTICES	EDGES		IDFS	IBFS	LT	SLT	SNCA	
idfsquad	1501	2500	28	2735.3	21.0	8.6	<b>4.2</b>	10.5	
ibfsquad	5004	10003	88	4.9	9519.4	8.8	4.5	<b>4.3</b>	
itworst	401	10501	34	6410.5	6236.8	9.2	<b>4.7</b>	<b>4.7</b>	
sltworst	32768	65534	2841	283.4	288.6	<b>7.9</b>	11.0	10.5	
sncaworst	10000	14999	179	523.2	243.8	12.1	<b>8.3</b>	360.7	
atis	4950	515080	2607	8.3	12.8	6.5	3.5	<b>3.3</b>	
nab	406555	939984	49048	17.6	15.6	12.8	11.6	<b>10.2</b>	
pw	330762	823330	42917	18.3	15.1	13.3	12.1	<b>10.4</b>	

We end our experimental analysis with results on artificial graphs. For each graph, Table 4 shows the number of vertices and edges, the time for BFS (in microseconds), and times for computing dominators (as multiples of BFS). The first five entries represent the worst-case families described in Section 3. The last three graphs have no special adversarial structure, but are significantly larger

than other graphs. As previously observed, the performance of iterative methods tends to degrade more noticeably with size. SNCA and SLT remain the fastest methods, but now LT comes relatively close. Given enough vertices, the asymptotically better behavior of LT starts to show.

## 5 Final Remarks

We compared five algorithms for computing dominators. Results on three classes of application graphs (program flow, VLSI circuits, and speech recognition) indicate that they have similar overall performance in practice. The tree-based iterative algorithms are the easiest to code and use less memory than the other methods, which makes them perform particularly well on small, simple graphs. Even on such instances, however, we did not observe the clear superiority of the original tree-based algorithm reported by Cooper et al. (our variants were not consistently better either). Both versions of LT and the hybrid algorithm are more robust on application graphs, and the advantage increases with graph size or graph complexity. Among these three, the sophisticated version of LT was the slowest, in contrast with the results reported by Lengauer and Tarjan [17]. The simple version of LT and hybrid were the most consistently fast algorithms in practice; since the former is less sensitive to pathological instances, it should be the method of choice.

**Acknowledgements.** We thank Adam Buchsbaum for providing us the SUIF and speech recognition graphs and Matthew Bridges for his help with IMPACT. We also thank the anonymous referees for their helpful comments.

## References

1. The IMPACT compiler. <http://www.crhc.uiuc.edu/IMPACT>.
2. The Standard Performance Evaluation Corp. <http://www.spec.org/>.
3. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
4. A. V. Aho and J. D. Ullman. *Principles of Compilers Design*. Addison-Wesley, 1977.
5. F. E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report IBM Res. Rep. RC 3923, IBM T.J. Watson Research Center, 1972.
6. S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.
7. M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana. Fault equivalence identification using redundancy information and static and dynamic extraction. In *Proceedings of the 19th IEEE VLSI Test Symposium*, March 2001.
8. A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–96, 1998. Corrigendum to appear.



9. K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Available online at <http://www.cs.rice.edu/~keith/EMBED/dom.pdf>.
10. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
11. H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 434–443, 1990.
12. L. Georgiadis and R. E. Tarjan. Finding dominators revisited. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms*, pages 862–871, 2004.
13. M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.
14. G. Holloway and C. Young. The flow analysis and transformation libraries of Machine SUIF. In *Proceedings of the 2nd SUIF Compiler Workshop*, 1997.
15. J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23:158–171, 1976.
16. D. E. Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, 1:105–133, 1971.
17. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flow-graph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–41, 1979.
18. P. W. Purdom, Jr. and E. F. Moore. Algorithm 430: Immediate predominators in a directed graph. *Communications of the ACM*, 15(8):777–778, 1972.
19. G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–296, 1994.
20. M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. volume 5, pages 141–153, 1980.
21. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.
22. P. H. Sweany and S. J. Beaty. Dominator-path scheduling: A global scheduling method. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 260–263, 1992.
23. R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.
24. R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.
25. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–81, 1984.
26. The CAD Benchmarking Lab, North Carolina State University. ISCAS’89 benchmark information. <http://www.cbl.ncsu.edu/www/CBL.Docs/iscas89.html>.

# Data Migration on Parallel Disks\*

Leana Golubchik<sup>1</sup>, Samir Khuller<sup>2</sup>, Yoo-Ah Kim<sup>2</sup>, Svetlana Shargorodskaya\*\*, and  
Yung-Chun (Justin) Wan<sup>2</sup>

<sup>1</sup> CS and EE-Systems Departments, IMSC, and ISI, University of Southern California,  
leana@cs.usc.edu

<sup>2</sup> Department of Computer Science and Institute for Advanced Computer Studies,  
University of Maryland, College Park, MD 20742,  
{samir,ykim,ycwan}@cs.umd.edu

**Abstract.** Our work is motivated by the problem of managing data on storage devices, typically a set of disks. Such storage servers are used as web servers or multimedia servers, for handling high demand for data. As the system is running, it needs to dynamically respond to changes in demand for different data items. There are known algorithms for mapping demand to a layout. When the demand changes, a new layout is computed. In this work we study the *data migration problem*, which arises when we need to quickly change one layout to another. This problem has been studied earlier when for each disk the new layout has been prescribed. However, to apply these algorithms effectively, we identify another problem that we refer to as the correspondence problem, whose solution has a significant impact on the solution for the data migration problem. We study algorithms for the data migration problem in more detail and identify variations of the basic algorithm that seem to improve performance in practice, even though some of the variations have poor worst case behavior.

## 1 Introduction

To handle high demand, especially for multimedia data, a common approach is to replicate data objects within the storage system. Typically, a large storage server consists of several disks connected using a dedicated network, called a *Storage Area Network*. Disks typically have constraints on storage as well as the number of clients that can access data from a single disk simultaneously. The goal is to have the system automatically respond to changes in demand patterns and to recompute data layouts. Such systems and their applications are described and studied in, e.g., [3,8] and the references therein.

Approximation algorithms have been developed [7,4] to map known demand for data to a specific data layout pattern to maximize utilization. (Utilization refers to the total number of clients that can be assigned to a disk that contains the data they want.) In the layout, we compute not only how many copies of each item we need, but also a layout

---

\* This research was supported by NSF Award CCR-0113192 and ANI-0070016. This work made use of Integrated Media Systems Center Shared Facilities supported by the National Science Foundation under Cooperative Agreement No. EEC-9529152; any Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation. Full version of this paper is available at: [http://www.cs.umd.edu/projects/smart/papers/esa04\\_full.ps](http://www.cs.umd.edu/projects/smart/papers/esa04_full.ps)

\*\* This work was done while this author was at the University of Maryland.

pattern that specifies the precise subset of items on each disk. (This is not completely accurate and we will elaborate on this step later.) The problem is *NP*-hard, but there are polynomial time approximation schemes [4]. Given the relative demand for data, an almost optimal layout can be computed.

Over time as the demand pattern changes, the system needs to create *new* data layouts. The problem we are interested in is the problem of computing a data migration plan for the set of disks to convert an initial layout to a target layout. We assume that data objects have the same size (these could be data blocks or files) and that it takes the same amount of time to migrate a data item between any pair of disks. The crucial constraint is that each disk can participate in the transfer of only one item – either as a sender or as a receiver. Our goal is to find a migration schedule to minimize the time taken (i.e., number of rounds) to complete the migration (makespan) since the system is running inefficiently until the new layout has been obtained.

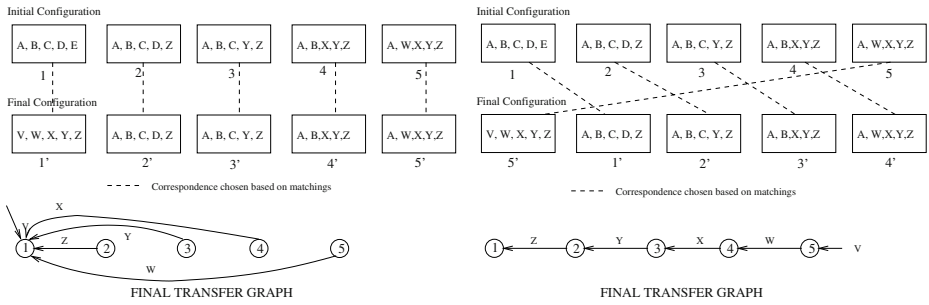
To handle high demand for newly popular objects, new copies will have to be dynamically created and stored on different disks. This means that we crucially need the ability to have a “copy” operation in addition to “move” operations. In fact, one of the crucial lower bounds used in the work on data migration [5] is based on a degree property of the multigraph. For example, if the degree of a node is  $\delta$ , then this is a lower bound on the number of rounds that are required, since in each round at most one transfer operation involving this node may be done. For copying operations, clearly this lower bound is not valid. For example, suppose we have a single copy of a data item on a disk. Suppose we wish to create  $\delta$  copies of this data item on  $\delta$  distinct disks. Using the transfer graph approach, we could specify a “copy” operation from the source disk to each of the  $\delta$  disks. Notice that this would take at least  $\delta$  rounds. However, by using newly created copies as additional sources we can create  $\delta$  copies in  $\lceil \log(\delta + 1) \rceil$  rounds, as in the classic problem of broadcasting by using newly created copies as sources for the data object. (Each copy spawns a new copy in each round.)

The *most general problem* of interest is the **data migration problem with cloning** [6] where data item  $i$  resides in a specified (source) subset  $S_i$  of disks and needs to be moved to a (destination) subset  $D_i$ . In other words, each data item that initially belongs to a subset of disks, needs to be moved to another subset of disks. (We might need to create new copies of this data item and store them on an additional set of disks.)

### 1.1 The Correspondence Problem

Given a set of data objects placed on disks, we shall assume that what is important is the grouping of the data objects and not their exact location on each disk. For example, we can represent a disk by the set  $\{A, B, C\}$  indicating that data objects  $A, B$ , and  $C$  are stored on this disk. If we move the location of these objects on the same disk, it does not affect the set corresponding to the disk in any way.

Data layout algorithms (such as the ones in [7,4]) take as input a demand distribution for a set of data objects and outputs a grouping  $S_{1'}, S_{2'}, \dots, S_{N'}$  as a desired data layout pattern on disks  $1', 2', \dots, N'$ . (The current layout is assumed to be  $S_1, S_2 \dots S_N$ .) Note that we do not need the data corresponding to the set of items  $S_{1'}$  to be on (original) disk 1. For example the algorithm simply requires that a new grouping be obtained where the items in set  $S_{1'}$  be grouped together on a disk. For instance, if  $S_3 = S_{1'}$  then by simply “renaming” disk 3 as disk  $1'$  we have obtained a disk with the set of items



**Fig. 1.** A bad correspondence can yield a poor solution (the left figure), while a good correspondence can yield significantly better solutions for data movement (the right figure).

$S_{1'}$ , assuming that these two disks are inter-changeable. We need to compute a perfect matching between the initial and final layout sets. An edge is present between  $S_i$  and  $S_{j'}$  if disk  $i$  has the same capabilities as disk  $j'$ . The weight of this edge is obtained by the number of “new items” that need to be moved to  $S_i$  to obtain  $S_{j'}$ . A minimum weight perfect matching in this graph gives the *correspondence* that minimizes the total number of changes, but *not* the number of rounds. Once we fix the correspondence, we need to invoke an algorithm to compute a migration schedule to minimize the number of rounds. Since this step involves solving an NP-hard problem, we will use a polynomial time approximation algorithm for computing the migration. However, we still need to pick a certain correspondence before we can invoke a data migration algorithm.

There are two central questions in which we are interested; these will be answered in Section 5:

- **Which correspondence algorithm should we use?** We will explore algorithms based on computing a matching of minimum total weight and matchings where we minimize the weight of the maximum weight edge. Moreover, the weight function will be based on estimates of how easy or difficult it is to obtain copies of certain data.
- **How good are our data migration algorithms once we fix a certain correspondence?** Even though we have bounds on the worst case performance of the algorithm, we would like to find whether or not its performance is a lot better than the worst case bound. (We do not have any example showing that the bound is tight.) In fact, it is possible that other heuristics perform extremely well, even though they do not have good worst case bounds [6].

For example, in the left figure of Figure 1 we illustrate a situation where we have 5 disks with the initial and final configurations as shown. By picking the correspondence as shown, we end up with a situation where all the data on the first disk needs to be changed. We have shown the possible edges that can be chosen in the transfer graph along with the labels indicating the data items that we could choose to transfer from the source disk to the destination disk. The final transfer graph shown is a possible output of a data migration algorithm. This will take 5 rounds since all the data is coming to a single disk; node 1 will have a high in-degree. Item  $V$  can be obtained from tertiary storage, for example (or another device). Clearly, this set of copy operations will be slow and will take many rounds.

On the other hand, if we use the correspondence as shown by the dashed edges in the right figure of Figure 1, we obtain a transfer graph where each disk needs only one new data item and such a transfer can be achieved in two rounds in parallel. (The set of transfers performed by the data migration algorithm are shown.)

## 1.2 Contributions

In recent work [6], it was shown that the data migration with cloning problem is NP-hard and has a polynomial time approximation algorithm (*KKW*) with a worst case guarantee of 9.5. Moreover, the work also explored a few simple data migration algorithms. Some of the algorithms cannot provide constant approximation guarantee, while for some of the algorithms no approximation guarantee is known. In this paper, we conduct an extensive study of these data migration algorithms' performance under different changes in user access patterns. We found that although the worst-case performance of the *KKW* algorithm is 9.5, in the experiments the number of rounds required is less than 3.25 times the lower bound. We identify the correspondence problem, and show that a good correspondence solution can improve the performance of the data migration algorithms by a factor of 4.4, relative to a bad solution. More detailed observations and results of the study are given in Section 5.

## 2 Models and Definitions

In the *data migration problem*, we have  $N$  disks and  $\Delta$  data items. For each item  $i$ , there is a subset of disks  $S_i$  and  $D_i$ . Initially only the disks in  $S_i$  have item  $i$ , and all disks in  $D_i$  want to receive  $i$ . Note that after a disk in  $D_i$  receives item  $i$ , it can be a source of item  $i$  for other disks in  $D_i$  which have not received the item yet. Our goal is to find the minimum number of rounds. Our algorithms make use of known results on edge coloring of multigraphs. Given a graph  $G$  with max degree  $\Delta_G$  and multiplicity  $\mu$ , it is well-known that it can be colored with at most  $\Delta_G + \mu$  colors.

## 3 Correspondence Problem Algorithms

To match disks in the initial layout with disks in the target layout, we tried the following methods, after creating a bipartite graph with two copies of disks:

1. (*Simple min max matching*) The weight of matching disk  $p$  in the initial layout with disk  $q$  in the target layout is the number of new items that disk  $q$  needs to get from other disks (because disks  $p$  does not have these items). Find a perfect matching that minimizes the maximum weight of the edges in the matching. Effectively it pairs disks in the initial layout to disks in the target layout, such that the number of items a disk needs to receive is minimized.
2. (*Simple min sum matching*) Minimum weighted perfect matching using the weight function defined in 1. This method minimizes the total number of transfer operations.
3. (*Complex min sum matching*) Minimum weighted perfect matching with another weight function that takes the ease of obtaining an item into account. Note that the larger the ratio of  $|D_i|$  to  $|S_i|$  the more copying is required. Suppose disk  $p$  in the

initial layout is matched with disk  $q$  in the target layout, and let  $S$  be the set of items that disk  $q$  needs which are not on disk  $p$ . The weight corresponding to matching these two disks is  $\sum_{i \in S} \max(\log \frac{|D_i|}{|S_i|}, 1)$ .

4. Direct correspondence. Disk  $i$  in the initial layout is always matched with disk  $i$  in the target layout.
5. Random permutation.

## 4 Data Migration Algorithms

### 4.1 KKW Data Migration Algorithm [6]

We now describe KKW data migration algorithm. Define  $\beta_j$  as  $|\{i | j \in D_i\}|$ , i.e., the number of different sets  $D_i$  to which a disk  $j$  belongs. We then define  $\beta$  as  $\max_{j=1 \dots N} \beta_j$ . In other words,  $\beta$  is an upper bound on the number of items a disk may need. Moreover, we may assume that  $D_i \neq \emptyset$  and  $D_i \cap S_i = \emptyset$ . (We simply define the destination set  $D_i$  as the set of disks that need item  $i$  and do not currently have it. Here we only give a high level description of the algorithm. The details can be found in a paper by Khuller, Kim, and Wan [6].

1. (*Choose sources*) For an item  $i$  decide a unique source  $s_i \in S_i$  so that  $\alpha = \max_{j=1, \dots, N} (|\{i | j = s_i\}| + \beta_j)$  is minimized. In other words,  $\alpha$  is the maximum number of items for which a disk may be a source ( $s_i$ ) or a destination.
2. (*Large destination sets*) Find a transfer graph for items such that  $|D_i| \geq \beta$ .
  - a) (*Choose representatives*) We first compute a disjoint collection of subsets  $G_i$ ,  $i = 1 \dots \Delta$ . Moreover,  $G_i \subseteq D_i$  and  $|G_i| = \lfloor \frac{|D_i|}{\beta} \rfloor$ .
  - b) (*Send to representatives*) We have each item  $i$  sent to the set  $G_i$ .
  - c) (*From representatives to destination sets*) We now create a transfer graph as follows. Each disk is a node in the graph. We add directed edges from disks in  $G_i$  to  $(\beta - 1) \lfloor \frac{|D_i|}{\beta} \rfloor$  disks in  $D_i \setminus G_i$  such that the out-degree of each node in  $G_i$  is at most  $\beta - 1$  and the in-degree of each node in  $D_i \setminus G_i$  is 1. We redefine  $D_i$  as a set of  $|D_i \setminus G_i| - (\beta - 1) \lfloor \frac{|D_i|}{\beta} \rfloor$  disks which do not receive item  $i$  so that they can be taken care of in Step 3. The redefined set  $D_i$  has size  $< \beta$ .
3. (*Small destination sets*) Find a transfer graph for items such that  $|D_i| < \beta$ .
  - a) (*Find new sources in small sets*) For each item  $i$ , find a new source  $s'_i$  in  $D_i$ . A disk  $j$  can be a source  $s'_i$  for several items as long as  $\sum_{i \in I_j} |D_i| \leq 2\beta - 1$  where  $I_j$  is a set of items of which  $j$  is a new source.
  - b) Send each item  $i$  from  $s_i$  to  $s'_i$ .
  - c) Create a transfer graph. We add a directed edge from the new source of item  $i$  to all disks in  $D_i \setminus \{s'_i\}$ .
4. We now find an edge coloring of the transfer graph obtained by merging two transfer graphs in Steps 2(c) and 3(c). The number of colors used is an upper bound on the number of rounds required to ensure that each disk in  $D_j$  receives item  $j$ .

**Theorem 1.** (Khuller, Kim, and Wan [6]) *The KKW algorithm described above has a worst-case approximation ratio of 9.5.*

## 4.2 KKW Data Migration Algorithm Variants

The 9.5-approximation algorithm for data migration [KKW (Basic)] uses several complicated components to achieve the constant factor approximation guarantee. We consider simpler variants of these components. The variants may not give good theoretical bounds.

- (a) In Step 2(a) (*Choose representatives*) we find the minimum integer  $m$  such that there exist disjoint sets  $G_i$  of size  $\lfloor \frac{|D_i|}{m} \rfloor$ . The value of  $m$  should be between  $\bar{\beta} = \sum_{i=1}^N \frac{\beta_i}{N}$  and  $\beta$ .
- (b) In Step 2(b) (*Send to representatives*) we use a simple doubling method to satisfy all requests in  $G_i$ . Since all groups are disjoint, it takes  $\max_i \log |G_i|$  rounds.
- (c) In Step 3 (*Small destination sets*) we do not find a new source  $s'_i$ . Instead  $s_i$  sends item  $i$  to  $D_i$  directly for small sets. We try to find a schedule which minimizes the maximum total degree of disks in the final transfer graph in Step 4.
- (d) In Step 3(a) (*Find new sources in small sets*) when we find a new source  $s'_i$ ,  $S_i$ s can be candidates as well as  $D_i$ s. If  $s'_i \in S_i$ , then we can save some rounds to send item  $i$  from  $s_i$  to  $s'_i$ .

## 4.3 Heuristic-Based Data Migration Algorithms [6]

1. [*Edge Coloring*] We can find a transfer graph, given the initial and target layouts. In the transfer graph, we have a set of nodes for disks and there is an edge between node  $j$  and  $k$  if for each item  $i$ ,  $j \in S_i$  and  $k \in D_i$ . Then we find an edge coloring of the transfer graph to obtain a valid schedule, where the number of colors used is an upper bound on the total number of rounds.
2. [*Unweighted Matching*] Find an unweighted matching in the transfer graph, schedule them, then create a new transfer graph based on the new  $S_i$  and  $D_i$ , and repeat.
3. [*Weighted Matching*] Repeatedly find and remove a weighted matching from the (repeatedly re-created) transfer graph, where the weight between disks  $v$  and  $w$  is  $\max_i (1 + \log \frac{|D_i|}{|S_i|})$ , over all items  $i$  where  $v \in S_i$ ,  $w \in D_i$ , or  $w \in S_i$ ,  $v \in D_i$ .
4. [*Item by Item*] We process each item  $i$  sequentially and satisfy the demand by doubling the number of copies of an item in each round.

## 5 Experiments

The framework of our experiments is as follows:

1. (*Create an initial layout*) Run the sliding window algorithm [4], given the number of user requests for each data object. In Section 5 we describe the distributions we used in generating user requests. These distributions are completely specified once we fix the ordering of data objects in order of decreasing demand.
2. (*Create a target layout*) Shuffle the ranking of items. Generate the new demand for each item according to the probabilities corresponding to the new ranking of the item. To obtain a target layout, take one of the following approaches.
  - a) Run the sliding window algorithm again with the new request demands.



- b) Use other (than sliding window) methods to create a target layout. The motivation for exploring these methods is (a) performance issues (as explained later in the paper) as well as (b) that other algorithms (other than sliding window) could perform well in creating layouts. The methods considered here are as follows.
  - i. Rotation of items: Suppose we numbered the items in non-increasing order of the number of copies in the initial layout. We make a sorted list of items of size  $k = \lfloor \frac{\Delta}{50} \rfloor$ , and let the list be  $l_1, l_2, \dots, l_k$ . Item  $l_i$  in the target layout will occupy the position of item  $l_{i+1}$  in the initial layout, while item  $l_k$  in the target layout will occupy the positions of item  $l_1$  in the initial layout. In other words, the number of copies of items  $l_1, \dots, l_{k-1}$  are decreased slightly, while the number of copies of item  $l_k$  is increased significantly.
  - ii. Enlarging  $D_i$  for items with small  $S_i$ : Repeat the following  $\lfloor \frac{\Delta}{20} \rfloor$  times. Pick an item  $s$  randomly having only one copy in the current layout. For each item  $i$  that has more than one copy in the current layout, there is a probability of 0.5 that item  $i$  will randomly give up the space of one of its copies, and the space will be allocated to item  $s$  in the new layout for the next iteration. In other words, if there are  $k$  items having more than one copy at the beginning of this iteration, then item  $s$  is expected to gain  $\frac{k}{2}$  copies at the end of the iteration.
3. (*Find a correspondence*) Run different correspondence algorithms given in Section 3 to match a disk in the initial layout with a disk in the target layout. Now we can find the set of source and destination disks for each item.
4. (*Compute a migration schedule*) Run different data migration algorithms, and record the number of rounds needed to finish the migration.

**User Request Distributions.** We generate the number of requests for different data objects using Zipf distributions and Geometric distributions. We note that few large-scale measurement studies exist for the applications of interest here (e.g., video-on-demand systems), and hence below we are considering several potentially interesting distributions. Some of these correspond to existing measurement studies (as noted below) and others we consider to explore the performance characteristics of our algorithms and to further improve the understanding of such algorithms. For instance, a Zipf distribution is often used for characterizing people's preferences.

*Zipf Distribution.* The Zipf distribution is defined as follows:  $\text{Prob}(\text{request for movie } i) = \frac{c}{i^{1-\theta}}, \forall i = 1, \dots, M$  and  $0 \leq \theta \leq 1$ , where  $c = \frac{1}{H_M^{1-\theta}}$ ,  $H_M^{1-\theta} = \sum_{j=1}^M \frac{1}{j^{1-\theta}}$ , and  $\theta$  determines the degree of skewness. We assign  $\theta$  to be 0 (similar to the *measurements* performed in [2]) and 0.5 in our experiments below.

*Geometric Distribution.* We also tried Geometric Distributions in order to investigate how a more skewed distribution affects the performance of the data migration algorithm. The distribution is defined as follows:  $\text{Prob}(\text{request for movie } i) = (1-p)^{i-1}p, \forall i = 1, \dots, M$  and  $0 < p < 1$ , where we use  $p$  set to 0.25 and 0.5 in our experiments below.

**Shuffling methods.** We use the following shuffling methods in Step 2(a) above.

1. Randomly promote 20% of the items. For each chosen item of rank  $i$ , we promote it to rank 1 to  $i-1$ , randomly.
2. Promote the least popular item to the top, and demote all other items by one rank.



**Parameters and Layout Creation.** We now describe the parameters used in the experiments, namely the number of disks, space capacity, and load capacity. We ran a number of experiments with 60 disks. For each correspondence method, user request distribution, and shuffling method, we generated 20 inputs (i.e., 20 sets of initial and target layouts) for each set of parameters, and ran different data migration algorithms on those instances. In the Zipf distribution, we used  $\theta$  values of 0 and 0.5, and in the Geometric distribution, we assigned  $p$  values of 0.25 and 0.5.

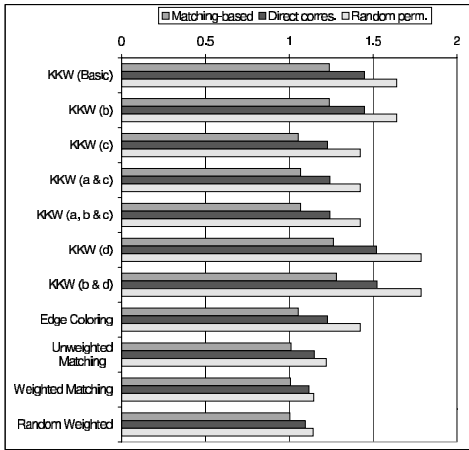
We tried three different pairs of settings for space and load capacities, namely: (A) 15 and 40, (B) 30 and 35, and (C) 60 and 150. We obtained these numbers from the specifications of modern SCSI hard drives. We show the results of 5 different layout creation schemes with different combinations of methods and parameters to create the initial and target layouts. (I): Promoting the last item to the top, Zipf distribution ( $\theta = 0$ ); (II): Promoting 20% of items, Zipf distribution ( $\theta = 0$ ); (III): Promoting the last item to the top, Geometric distribution ( $p = 0.5$ ); (IV): Initial layout obtained from the Zipf distribution ( $\theta = 0$ ), target layout obtained from the method described in Step 2(b)i in Section 5 (rotation of items); and (V): Initial layout obtained from the Zipf distribution ( $\theta = 0$ ), target layout obtained from the method described in Step 2(b)ii in Section 5 (enlarging  $D_i$  for items with small  $S_i$ ). We also tried promoting 20% of items with Geometric distribution. The result is omitted since it gave qualitatively similar results.

## Results

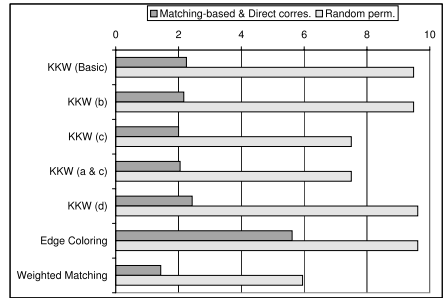
In the tables we present the average for 20 inputs. Moreover, we present results of two representative inputs individually, to illustrate the performance of the algorithms under the same initial and target layouts. This presentation is motivated as follows. The *absolute* performance of each run is largely a function of the differences between the initial and the target layouts (and this is true for all algorithms). That is, a small difference is likely to result in relatively few rounds needed for data migration, and a large difference is likely to result in relatively many rounds needed for data migration. Since a goal of this study is to understand the *relative* performance differences between the algorithms described above, i.e., given the same initial and target layouts, we believe that presenting the data on a per run basis is more informative. That is, considering the average alone somewhat obscures the characteristics of the different algorithms.

**Different correspondence methods.** We first investigate how different correspondence methods affect the performance of the data migration algorithms. Figures 2 and 3 show the ratio of the number of rounds taken by different data migration algorithms to the lower bounds, averaged over 20 inputs under parameter setting (A). We observed that the simple min max matching (1) always returns the same matching as the simple min sum matching (2) in all instances we tried. Moreover, using a simpler weight function (2) or a more involved one (3) does not seem to affect the behavior in any significant way (often these matchings are the same). Thus we only present results using simple min max matching, and label this as matching-based correspondence method.

From the figures, we found that using a matching-based method is important and can affect the performance of all algorithms by up to a factor of 4.4 as compared to a bad correspondence, using a random permutation for example. Since direct correspondence does not perform as well as other weight-based matchings, this also suggests that



**Fig. 2.** The ratio of the number of rounds taken by the algorithms to the lower bound (28.1 rounds), averaged over 20 inputs, using parameter setting (A) and layout creation scheme (I).



**Fig. 3.** The ratio of the number of rounds taken by the algorithms to the lower bound (6.0 rounds), averaged over 20 inputs, using parameter setting (A) and layout creation scheme (II).

a good correspondence method is important. However, the performance of direct correspondence was reasonable when we promoted the popularity of one item. This can be explained by the fact that in this case sliding window obtains a target layout which is fairly similar to the initial layout.

**Different data migration algorithms.** From the previous section it is reasonable to evaluate the performance of different data migration algorithms using only the simple min max matching correspondence method. We now compare the performance of different variants of the KKW algorithm. Consider algorithm KKW (c) which modifies Step 3 (where we want to satisfy small  $D_i$ ); we found that sending the items from  $S_i$  to small  $D_i$  directly using edge coloring, without using new sources  $s'_i$ , is a much better idea. Even though this makes the algorithm an  $O(\Delta)$  approximation algorithm, the performance is very good under both the Zipf and the Geometric distributions, since the sources are not concentrated on one disk and only a few items require rapid doubling.

In addition, we thought that making the sets  $G_i$  slightly larger by using  $\bar{\beta}$  was a better idea (i.e., algorithm KKW (a) which modifies Step 2(a)). This reduces the average degree of nodes in later steps such as in Step 2(c) and Step 3 where we send the item to disks in  $D_i \setminus G_i$ . However, the experiments shown it usually performs slightly worse than the algorithm using  $\beta$ .

Consider algorithm KKW (d) which modifies Step 3(a) (where we identify new sources  $s'_i$ ); we found that the performance of the variant that includes  $S_i$ , in addition to  $D_i$ , as candidates for the new source  $s'_i$  is mixed. Sometimes it is better than the basic KKW algorithm, but more often it is worse.

Consider algorithm KKW (b) which modifies Step 2(b) (where we send the items from the sources  $S_i$  to  $G_i$ ); we found that doing a simple broadcast is generally a better

**Table 1.** The ratio of the number of rounds taken by the data migration algorithms to the lower bounds, using min max matching correspondence method with layout creation scheme (III) (i.e., promoting the last item to the top, with user requests following the Geometric distribution ( $p = 0.5$ )), and under different parameter settings.

Parameter setting: Instance:	(A)			(B)		
	1	2	Ave	1	2	Ave
KKW (Basic)	2.000	2.167	2.250	1.611	1.611	1.568
KKW (b)	1.875	2.167	2.167	1.611	1.611	1.568
KKW (c)	1.625	2.000	2.000	1.444	1.222	1.286
KKW (a & c)	1.750	2.000	2.050	1.333	1.333	1.296
KKW (a, b & c)	1.625	2.000	1.917	1.278	1.278	1.261
KKW (d)	1.750	2.333	2.433	1.722	1.500	1.533
KKW (b & d)	1.875	2.333	2.483	1.556	1.556	1.538
Edge Coloring	3.875	5.667	5.617	2.000	2.111	1.980
Unweighted Matching	1.000	1.500	1.500	1.111	1.111	1.116
Weighted Matching	1.000	1.500	1.433	1.056	1.111	1.111
Random Weighted	1.000	1.333	1.367	1.056	1.056	1.010
Item by Item	6.250	12.833	12.683	3.667	3.667	5.719

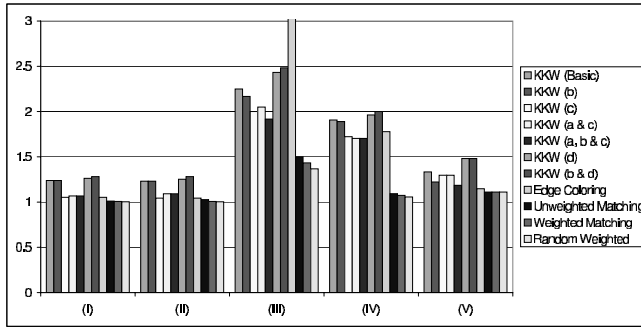
**Table 2.** The ratio of the number of rounds taken by the data migration algorithms to the lower bounds, using min max matching correspondence method with layout creation scheme (IV) (i.e., target layout obtained from the method described in Step 2(b)i in Section 5 (rotation of items)), and under different parameter settings.

Parameter setting: Instance:	(A)			(B)			(C)		
	1	2	Ave	1	2	Ave	1	2	Ave
KKW (Basic)	2.400	2.000	1.907	1.417	1.444	1.553	1.333	1.250	1.330
KKW (b)	2.200	2.000	1.889	1.417	1.444	1.553	1.333	1.250	1.330
KKW (c)	2.000	1.600	1.722	1.167	1.111	1.289	1.222	1.000	1.107
KKW (a & c)	1.800	2.000	1.704	1.250	1.333	1.408	1.222	1.083	1.170
KKW (a, b & c)	2.000	2.000	1.704	1.333	1.333	1.408	1.222	1.083	1.170
KKW (d)	2.400	2.400	1.963	1.333	1.444	1.513	1.556	1.083	1.348
KKW (b & d)	2.200	2.200	2.000	1.250	1.667	1.658	1.556	1.333	1.393
Edge Coloring	1.800	2.000	1.778	1.000	1.000	1.250	1.222	1.000	1.125
Unweighted Matching	1.000	1.000	1.093	1.000	1.000	1.026	1.000	1.000	1.000
Weighted Matching	1.000	1.200	1.074	1.000	1.000	1.013	1.000	1.000	1.009
Random Weighted	1.000	1.200	1.056	1.000	1.000	1.013	1.000	1.000	1.009
Item by Item	10.000	9.800	8.889	6.333	8.111	9.592	15.778	12.000	12.536

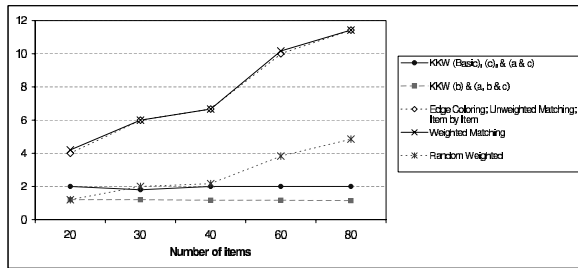
idea, as we can see from the results for Parameter Setting (A), Instance 1 in Table 1 and for Parameter Setting (A), Instance 1 in Table 2.

Even though this makes the algorithm an  $O(\log n)$  approximation algorithm, very rarely is the size of  $\max G_i$  large. Under the input generated by Zipf and Geometric distributions, the simple broadcast generally performs the same as the basic KKW algorithm since the size of  $\max G_i$  is very small.

Out of all the heuristics, the matching-based heuristics perform very well in practice. The only cases where they perform extremely badly correspond to hand-crafted (by us) bad examples. Suppose, for example, a set of  $\Delta$  disks are the sources for  $\Delta$  items (each disk has all  $\Delta$  items). Suppose further that the destination disks also have size  $\Delta$  each and are disjoint. The result is listed in Figure 5. The KKW algorithm sends each of the items to one disk in  $D_i$  in the very first round. After that, a broadcast can be done in the destination sets as they are disjoint, which takes  $O(\log \Delta)$  rounds in total. Therefore the ratio of the number of rounds used to the lower bound remains a constant. The matching-



**Fig. 4.** The ratio of the number of rounds taken by the algorithms to the lower bound, averaged over 20 inputs, using min max matching correspondence method and parameter setting (A), under different layout creation schemes (see Section 5). Note that the order of the bars in each cluster is the same as that in the legend.



**Fig. 5.** The ratio of the number of rounds taken by the algorithms to the lower bound under the worst case.

based algorithm can take up to  $\Delta$  rounds, as it can focus on sending item  $i$  at each round by computing a perfect matching of size  $\Delta$  between the source disks and the destination disks for item  $i$  in each round. Since any perfect matching costs the same weight in this case, the matching focuses on sending only one item in each round. We implemented a variant of the weighted matching heuristic to get around this problem by adding a very small random weight to each edge in the graph. As we can see from Figure 5, although this variant does not perform as well as the KKW algorithm, it performs much better than other matching-based data migration algorithms. Moreover, we ran this variant with the inputs generated from Zipf and Geometric distributions, and we found that it frequently takes the same number of rounds as the weighted matching heuristic. In some cases it performs better than the weighted matching heuristic, while in a few cases its performance is worse.

Given the performance of different data migration algorithms illustrated in Figure 4 and in the tables, the two matching-based heuristics are comparable. Matching-based heuristics perform the best in general, followed by the edge-coloring heuristic, followed by the KKW algorithms, while processing items one-by-one comes last. The main reason why the edge-coloring heuristic performs better than the basic KKW algorithm is that the

**Table 3.** The ratio of the number of rounds taken by the data migration algorithms to the lower bounds, using min max matching correspondence method with layout creation scheme (V) (i.e., target layout obtained from the method described in Step 2(b)ii in Section 5 (enlarging  $D_i$  for items with small  $S_i$ )), and under different parameter settings.

Parameter setting: Instance:	(A)			(B)			(C)		
	1	2	Ave	1	2	Ave	1	2	Ave
KKW (Basic)	1.000	1.333	1.333	1.000	1.000	1.114	1.167	1.000	1.140
KKW (b)	1.000	1.333	1.222	1.000	1.000	1.114	1.167	1.000	1.140
KKW (c)	1.000	1.333	1.296	1.000	1.000	1.086	1.000	1.000	1.093
KKW (a & c)	1.000	1.333	1.296	1.000	1.000	1.086	1.167	1.000	1.116
KKW (a, b & c)	1.000	1.333	1.185	1.000	1.000	1.086	1.167	1.000	1.093
KKW (d)	1.000	1.667	1.481	1.000	1.500	1.286	1.500	1.750	1.488
KKW (b & d)	1.000	1.667	1.481	1.000	1.500	1.286	1.667	1.750	1.512
Edge Coloring	1.000	1.000	1.148	1.000	1.000	1.057	1.000	1.000	1.047
Unweighted Matching	1.000	1.000	1.111	1.000	1.000	1.029	1.000	1.000	1.047
Weighted Matching	1.000	1.000	1.111	1.000	1.000	1.029	1.000	1.000	1.047
Random Weighted	1.000	1.000	1.111	1.000	1.000	1.029	1.000	1.000	1.047
Item by Item	7.000	5.000	5.815	15.000	7.750	8.200	8.833	11.750	11.349

input contains mostly move operations, i.e., the size of  $S_i$  and  $D_i$  is at most 2 for at least 80% of the items. The Zipf distribution does not provide enough cloning operations for the KKW algorithm to show its true potential (the sizes of  $G_i$  are mostly zero, with one or two items having size of 1 or 2). Processing items one by one is bad because we have a lot of items (more than 300 in Parameter Setting (A)), but most of the operations can be done in parallel (we have 60 disks, meaning that we can support 30 copy operations in one round). Under the Zipf distribution, since the sizes of most sets  $G_i$  are zero, the data migration variant which sends items from  $S_i$  directly to  $D_i$  is essentially the same as the coloring heuristic. Thus they have almost the same performance.

*Under different input parameters.* In addition to the Zipf distribution, we also tried the Geometric distribution because we would like to investigate the performance of the algorithms under more skewed distributions where more cloning of items is necessary. As we can see in Figure 3 and Table 1, we found that the performance of the coloring heuristic is worse than the KKW algorithms, especially when  $p$  is large (more skewed) or when the ratio of the load capacity to space capacity is high. However, the matching-based heuristics still perform the best.

We also investigated the effect on the performance of different algorithms under a higher ratio of the load to space capacities. We found that the results to be qualitatively similar, and thus we omit them here.

Moreover, in the Zipf distribution, we assigned different values of  $\theta$ , which controls the skewness of the distribution, as 0.0 and 0.5. We found that the results are similar in both cases. While in the Geometric distribution, a higher value of  $p$  (0.5 vs 0.25) gives the KKW algorithms an advantage over coloring heuristics as more cloning is necessary.

*Miscellaneous.* Tables 2 and 3 show the performance of different algorithms using inputs where the target layout is derived from the initial layout, as described in Step 2(b)i and Step 2(b)ii in Section 5. Note that when the initial and target layouts are very similar, the data migration can be done very quickly. The number of rounds taken is much fewer than the number of rounds taken using inputs generated from running the sliding window algorithm twice. This illustrates that it may be worthwhile to consider developing an algorithm which takes in an initial layout and the new access pattern, and then derives a

target layout, with the optimization of the data migration process in mind. Developing these types of algorithms and evaluating their performance characteristics is part of our future efforts.

We compared the performance of different algorithms under two shuffling methods described in Section 5. We found the results to be qualitative similar, and thus we omit them here due to lack of space.

We now consider the running time of the different data migration algorithms. Except for the matching heuristics, all other algorithms' running times are at most 3 CPU seconds and often less than 1 CPU second, on a Sun Fire V880 server. The running time of the matching heuristics depends on the total number of items. It can be as high as 43 CPU seconds when the number of items is around 3500, and it can be lower than 2 CPU seconds when the number of items is around 500.

**Final Conclusions.** For the correspondence problem question posed in Section 1.1, our experiments indicate that weighted matching is the best approach among the ones we tried.

For the data migration problem question posed in Section 1.1, our experiments indicate that the weighted matching heuristic with some randomness does very well. This suggests that perhaps a variation of matching can be used to obtain an  $O(1)$  approximation as well. Among all variants of the KKW algorithm, letting  $S_i$  send item  $i$  to  $D_i$  directly for the small sets, i.e., variant (c), performs the best. From the above described results we can conclude that under the Zipf and Geometric distributions, where cloning does not occur frequently, the weighted matching heuristic returns a schedule which requires only a few rounds more than the optimal. All variants of the KKW algorithm usually take no greater than 10 more rounds than the optimal, when a good correspondence method is used.

## References

1. E. Anderson, J. Hall, J. Hartline, M. Hobbes, A. Karlin, J. Saia, R. Swaminathan and J. Wilkes. An Experimental Study of Data Migration Algorithms. *Workshop on Algorithm Engineering*, 2001
2. A. L. Chervenak. Tertiary Storage: An Evaluation of New Applications. *Ph.D. Thesis, UC Berkeley*, 1994.
3. S. Ghandeharizadeh and R. R. Muntz. Design and Implementation of Scalable Continuous Media Servers. *Parallel Computing Journal*, 24(1):91–122, 1998.
4. L. Golubchik, S. Khanna, S. Khuller, R. Thurimella and A. Zhu. Approximation Algorithms for Data Placement on Parallel Disks. *Proc. of ACM-SIAM SODA*, 2000.
5. J. Hall, J. Hartline, A. Karlin, J. Saia and J. Wilkes. On Algorithms for Efficient Data Migration. *Proc. of ACM-SIAM SODA*, 620–629, 2001.
6. S. Khuller, Y. Kim and Y-C. Wan. Algorithms for Data Migration with Cloning. *22nd ACM Symposium on Principles of Database Systems (PODS)*, 27–36, 2003.
7. H. Shachnai and T. Tamir. On two class-constrained versions of the multiple knapsack problem. *Algorithmica*, 29:442–467, 2001.
8. J. Wolf, H. Shachnai and P. Yu. DASD Dancing: A Disk Load Balancing Optimization Scheme for Video-on-Demand Computer Systems. *ACM SIGMETRICS/Performance Conf.*, 157–166, 1995.

# Classroom Examples of Robustness Problems in Geometric Computations<sup>\*</sup>

## (Extended Abstract)

Lutz Kettner<sup>1</sup>, Kurt Mehlhorn<sup>1</sup>, Sylvain Pion<sup>2</sup>, Stefan Schirra<sup>3</sup>, and Chee Yap<sup>4</sup>

<sup>1</sup> MPI für Informatik, Saarbrücken, {kettner,mehlhorn}@mpi-sb.mpg.de

<sup>2</sup> INRIA Sophia Antipolis, Sylvain.Pion@sophia.inria.fr

<sup>3</sup> Otto-von-Guericke-Universität, Magdeburg, stschirr@isg.cs.uni-magdeburg.de

<sup>4</sup> New York University, New York, USA, yap@cs.nyu.edu

**Abstract.** The algorithms of computational geometry are designed for a machine model with exact real arithmetic. Substituting floating point arithmetic for the assumed real arithmetic may cause implementations to fail. Although this is well known, there is no comprehensive documentation of what can go wrong and why. In this extended abstract, we study a simple incremental algorithm for planar convex hulls and give examples which make the algorithm fail in all possible ways. We also show how to construct failure-examples semi-systematically and discuss the geometry of the floating point implementation of the orientation predicate. We hope that our work will be useful for teaching computational geometry. The full paper is available at [www.mpi-sb.mpg.de/~mehlhorn/ftp/ClassRoomExamples.ps](http://www.mpi-sb.mpg.de/~mehlhorn/ftp/ClassRoomExamples.ps). It contains further examples, more theory, and color pictures. We strongly recommend to read the full paper instead of this extended abstract.

## 1 Introduction

The algorithms of computational geometry are designed for a machine model with exact real arithmetic. Substituting floating point arithmetic for the assumed real arithmetic may cause implementations to fail. Although this is well known, it is not common knowledge. There is no paper which systematically discusses what can go wrong and provides simple examples for the different ways in which floating point implementations can fail. Due to this lack of examples, instructors of computational geometry have little material for demonstrating the inadequacy of floating point arithmetic for geometric computations, students of computational geometry and implementers of geometric algorithms still underestimate the seriousness of the problem, and researchers in our and neighboring disciplines, e.g., numerical analysis, still believe, that simple approaches are able to overcome the problem.

In this extended abstract, we study a simple incremental algorithm for planar convex hulls and show how it can fail and why it fails when executed with floating point arithmetic. The convex hull  $CH(S)$  of a set  $S$  of points in the plane is the smallest convex polygon containing  $S$ . A point  $p \in S$  is called *extreme* in  $S$  if  $CH(S) \neq CH(S \setminus p)$ .

---

<sup>\*</sup> Partially supported by the IST Programme of the EU under Contract No IST-2000-26473 (Effective Computational Geometry for Curves and Surfaces (ECG)).

The extreme points of  $S$  form the vertices of the convex hull polygon. Convex hulls can be constructed incrementally. One starts with three non-collinear points in  $S$  and then considers the remaining points in arbitrary order. When a point is considered and lies inside the current hull, the point is simply discarded. When the point lies outside, the tangents to the current hull are constructed and the hull is updated appropriately. We give a more detailed description of the algorithm in Section 4 and the complete C++ program in the full paper.

Figures 2 and 5 show point sets (we give the numerical coordinates of the points in Section 4) and the respective convex hulls computed by the floating point implementation of our algorithm. In each case the input points are indicated by small circles, the computed convex hull polygon is shown in light grey, and the alleged extreme points are shown as filled circles. The examples show that the implementation may make gross mistakes. It may leave out points which are clearly extreme, it may compute polygons which are clearly non-convex, and it may even run forever (example not shown here).

The first contribution of this paper is to provide a set of instances that make the floating point implementation fail in disastrous ways. The computed results do not resemble the correct results in any reasonable sense.

Our second contribution is to explain why these disasters happen. The correctness of geometric algorithms depends on geometric properties, e.g., a point lies outside a convex polygon if and only if it can see one of the edges. We give examples, for which a floating point implementation violates these properties: a point outside a convex polygon which sees no edge (in a floating point implementation of “sees”) and a point not outside which sees some edges (in a floating point implementation of “sees”). We give examples for all possible violations of the correctness properties of our algorithm.

Our third contribution is to show how difficult examples can be constructed systematically or at least semi-systematically. This should allow others to do similar studies.

We believe that the paper and its companion web page will be useful in teaching computational geometry, and that even experts will find it surprising and instructing in how many ways and how badly even simple algorithms can be made to fail. The companion web page (<http://www.mpi-sb.mpg.de/~kettner/proj/NonRobust/>) contains the source code of all programs, the input files for all examples, and installation procedures. It allows the reader to perform our and further experiments.

This paper is structured as follows. In Section 2 we discuss the ground rules for our experiments, in Section 3 we study the effect of floating point arithmetic on one of the most basic predicates of planar geometry, the orientation predicate, in Section 4 we discuss an incremental algorithm for planar convex hulls. Finally, Section 5 offers a short conclusion. In the full paper, we also discuss the gift wrapping algorithm for planar convex hulls and an incremental algorithm for 3d Delaunay triangulations, and also give additional theory.

*Related Work:* The literature contains a small number of documented failures due to numerical imprecision, e.g., Forrest’s seminal paper on implementing the point-in-polygon test [For85], Shewchuk’s example for divide-and-conquer Delaunay triangulation [She97], Ramshaw’s braided lines [MN99, Section 9.6.2], Schirra’s example for convex hulls [MN99, Section 9.6.1], and the sweep line algorithm for line segment intersection and boolean operations on polygons [MN99, Sections 10.7.4 and 10.8.3].



## 2 Ground Rules for Our Experiments

Our codes are written in C++ and the results are reproducible on any platform compliant with IEEE floating-point standard 754 for double precision (see [Gol90]), and also with other programming languages. All programs and input data can be found on the companion web page. Numerical computations are based on IEEE arithmetic. In particular, numbers are machine floating point numbers (called *doubles* for short). Such numbers have the form  $\pm m2^e$  where  $m = 1.m_1m_2 \dots m_{52}$  ( $m_i = \{0, 1\}$ ) is the mantissa in binary and  $e$  is the exponent satisfying  $-1024 < e < 1024$ . The results of arithmetic operations are rounded to the nearest double (with ties broken using some fixed rule).

Our numerical example data will be written in decimals (for human consumption). Such decimal values, when read into the machine, are internally represented by the nearest double. We have made sure that our data can be represented exactly by doubles and their conversion to binary and back to decimal is the identity operation.

## 3 The Orientation Predicate

Three points  $p$ ,  $q$ , and  $r$  in the plane either lie on a common line or form a left or right turn. The triple  $(p, q, r)$  forms a left (right) turn, if  $r$  lies left (right) of the line through  $p$  and  $q$  and oriented in the direction from  $p$  to  $q$ . Analytically, the orientation of the triple  $(p, q, r)$  is tantamount to the sign of a determinant:

$$\text{orientation}(p, q, r) = \text{sign}(\det \begin{bmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{bmatrix}), \quad (1)$$

where  $p = (p_x, p_y)$ , etc. We have  $\text{orientation}(p, q, r) = +1$  (resp.,  $-1, 0$ ) iff the polyline  $(p, q, r)$  represents a left turn (resp., right turn, collinearity). Interchanging two points in the triple changes the sign of the orientation. We denote the  $x$ -coordinate of a point  $p$  also by  $x(p)$ . We implement the orientation predicate in the straightforward way:

$$\text{orientation}(p, q, r) = \text{sign}((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)). \quad (2)$$

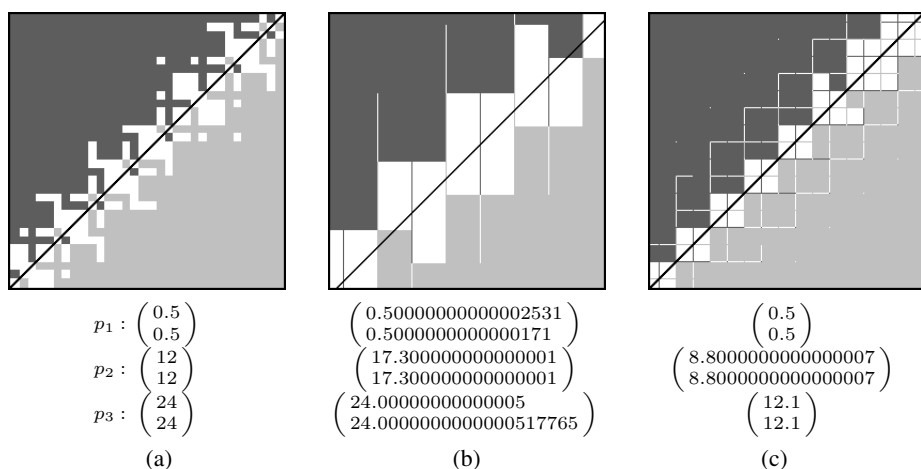
When the orientation predicate is implemented in this obvious way and evaluated with floating point arithmetic, we call it *float\_orient* $(p, q, r)$  to distinguish it from the ideal predicate. Since floating point arithmetic incurs round-off error, there are potentially three ways in which the result of *float\_orient* could differ from the correct orientation:

- *rounding to zero*: we mis-classify a  $+$  or  $-$  as a 0;
- *perturbed zero*: we mis-classify 0 as  $+$  or  $-$ ;
- *sign inversion*: we mis-classify a  $+$  as  $-$  or vice-versa.

### 3.1 The Geometry of Float-Orientation

What is the geometry of *float\_orient*, i.e., which triples of points are classified as left-turns, right-turns, or straight? The following type of experiment answers the question: We choose three points  $p_1$ ,  $p_2$ , and  $p_3$  and then compute

$$\text{float\_orient}((x(p_1) + x \cdot u, y(p_1) + y \cdot u), p_2, p_3)$$



**Fig. 1.** The weird geometry of the float-orientation predicate: The figure shows the results of  $\text{float\_orient}((x(p_1) + x \cdot u, y(p_1) + y \cdot u, p_2, p_3))$  for  $0 \leq x, y \leq 255$ , where  $u$  is the increment between adjacent floating point numbers in the considered range. The result is color coded: White (light grey, dark grey, resp.) pixels represent collinear (negative, positive, resp.) orientation. The line through  $p_2$  and  $p_3$  is also shown.

for  $0 \leq x, y \leq 255$ , where  $u$  is the increment between adjacent floating point numbers in the considered range; for example,  $u = 2^{-53}$  if  $x(p_1) = 1/2$  and  $u = 4 \cdot 2^{-53}$  if  $x(p_1) = 2 = 4 \cdot 1/2$ . We visualize the resulting  $256 \times 256$  array of signs as a  $256 \times 256$  grid of pixels: A white (light grey, dark grey) pixel represents collinear (negative, positive, respectively) orientation. In the figures in this section we also indicate an approximation of the line through  $p_2$  and  $p_3$ .

Figure 1(a) shows the result of our first experiment: We use the line defined by the points  $p_2 = (12, 12)$  and  $p_3 = (24, 24)$  and query it near  $p_1 = (0.5, 0.5)$ . We urge the reader to pause for a moment and to sketch what he/she expects to see. The authors expected to see a white band around the diagonal with nearly straight boundaries. Even for points with such simple coordinates the geometry of *float\_orient* is quite weird: the set of white points (= the points classified as on the line) does not resemble a straight line and the sets of light grey or dark grey points do not resemble half-spaces. We even have points that change the side of the line, i.e., are lying left (right) of the line and being classified as right (left) of the line.

In Figures 1(b) and (c) we have given our base points  $p_i$  more complex coordinates by adding some digits behind the binary point. This enhances the cancellation effects in the evaluation of *float\_orient* and leads to even more striking pictures. In (b), the light grey region looks like a step function at first sight. Note however, it is not monotone, has white rays extending into it, and light grey lines extruding from it. The white region (= on-region) forms blocks along the line. Strangely enough, these blocks are separated by dark and light grey lines. Finally, many points change sides. In Figure (c), we have white blocks of varying sizes along the diagonal, thin white and partly light grey lines

extending into the dark grey region (similarly for the light grey region), light grey points (the left upper corners of the white structures extending into the dark grey region) deep inside the blue region, and isolated white points almost 100 units away from the diagonal.

All diagrams in Figure 1 exhibit block structure. We now explain why. Assume we keep  $y$  fixed and vary only  $x$ . We evaluate  $\text{float\_orient}((x(p_1) + x \cdot u, y(p_1) + y \cdot u), p_2, p_3)$  for  $0 \leq x \leq 255$ , where  $u$  is the increment between adjacent floating point numbers in the considered range. Also  $\text{orientation}(p, q, r) = \text{sign}((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x))$ . We incur round-off error in the additions/subtractions and also in the multiplications. Consider first one of the differences, say  $q_x - p_x$ . In (a), we have  $q_x \approx 12$  and  $p_x \approx 0.5$ . Since 12 has four binary digits, we loose the last four bits of  $x$  in the subtraction, in other words, the result of the subtraction  $q_x - p_x$  is constant for  $2^4$  consecutive values of  $x$ . Because of rounding to nearest, the intervals of constant value are  $[8, 23]$ ,  $[24, 39]$ ,  $[40, 55]$  . . . . Similarly, the floating point result of  $r_x - p_x$  is constant for  $2^5$  consecutive values of  $x$ . Because of rounding to nearest, the intervals of constant value are  $[16, 47]$ ,  $[48, 69]$ , . . . . Overlaying the two progressions gives intervals  $[16, 23]$ ,  $[24, 39]$ ,  $[40, 47]$ ,  $[48, 55]$ , . . . and this explains the structure we see in the rows of (a). We see short blocks of length 8, 16, 24, . . . in (a). In (b) and (c), the situation is somewhat more complicated. It is again true that we have intervals for  $x$ , where the results of the subtractions are constant. However, since  $q$  and  $r$  have more complex coordinates, the relative shifts of these intervals are different and hence we see narrow and broad features.

## 4 The Convex Hull Problem

We discuss a simple incremental convex hull algorithm. We describe the algorithm, state the underlying geometric assumptions, give instances which violate the assumptions when used with floating point arithmetic, and finally show which disastrous effects these violations may have on the result of the computation.

The incremental algorithm maintains the *current convex hull*  $CH$  of the points seen so far. Initially,  $CH$  is formed by choosing three non-collinear points in  $S$ . It then considers the remaining points one by one. When considering a point  $r$ , it first determines whether  $r$  is outside the current convex hull polygon. If not,  $r$  is discarded. Otherwise, the hull is updated by forming the tangents from  $r$  to  $CH$  and updating  $CH$  appropriately. The incremental paradigm is used in Andrew's variant [And79] of Graham's scan [Gra72] and also in the randomized incremental algorithm [CS89].

The algorithm maintains the current hull as a circular list  $L = (v_0, v_1, \dots, v_{k-1})$  of its extreme points in counter-clockwise order. The line segments  $(v_i, v_{i+1})$ ,  $0 \leq i \leq k-1$  (indices are modulo  $k$ ) are the *edges* of the current hull. If  $\text{orientation}(v_i, v_{i+1}, r) < 0$ , we say  $r$  *sees* the edge  $(v_i, v_{i+1})$ , and say the edge  $(v_i, v_{i+1})$  is *visible* from  $r$ . If  $\text{orientation}(v_i, v_{i+1}, r) \leq 0$ , we say that the edge  $(v_i, v_{i+1})$  is *weakly visible* from  $r$ . After initialization,  $k \geq 3$ . The following properties are key to the operation of the algorithm.

**Property A:** A point  $r$  is outside  $CH$  iff  $r$  can see an edge of  $CH$ .

**Property B:** If  $r$  is outside  $CH$ , the edges weakly visible from  $r$  form a non-empty consecutive subchain; so do the edges that are not weakly visible from  $r$ .

If  $(v_i, v_{i+1}), \dots, (v_{j-1}, v_j)$  is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence  $(v_{i+1}, \dots, v_{j-1})$  by  $r$ . The subsequence  $(v_i, \dots, v_j)$  is taken in the circular sense. E.g., if  $i > j$  then the subsequence is  $(v_i, \dots, v_{k-1}, v_0, \dots, v_j)$ . From these properties, we derive the following algorithm:

```

Initialize  $L$  to the counter-clockwise triangle  $(a, b, c)$ . Remove  $a, b, c$  from  $S$ .
for all  $r \in S$  do
  if there is an edge  $e$  visible from  $r$  then
    Compute the sequence  $(v_i, \dots, v_j)$  of edges that are weakly visible from  $r$ ;
    Replace the subsequence  $(v_{i+1}, \dots, v_{j-1})$  by  $r$ ;
  end if
end for

```

To turn the sketch into an algorithm, we provide more information about the substeps.

1. How does one determine whether there is an edge visible from  $r$ ? We iterate over the edges in  $L$ , checking each edge using the orientation predicate. If no visible edge is found, we discard  $r$ . Otherwise, we take any one of the visible edges as the starting edge for the next item.
2. How does one identify the subsequence  $(v_i, \dots, v_j)$ ? Starting from a visible edge  $e$ , we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, move clockwise from  $e$  until a non-weakly-visible edge is encountered.
3. How to update the list  $L$ ? We can delete the vertices in  $(v_{i+1}, \dots, v_{j-1})$  after all visible edges are found, as suggested in the above sketch (“the off-line strategy”) or we can delete them concurrently with the search for weakly visible edges (“the on-line strategy”).

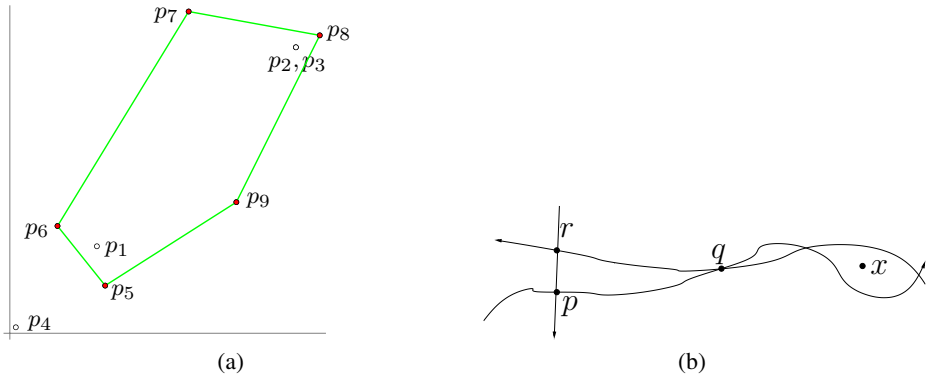
There are four logical ways to negate Properties A and B:

- **Failure (A<sub>1</sub>):** A point outside the current hull sees no edge of the current hull.
- **Failure (A<sub>2</sub>):** A point inside the current hull sees an edge of the current hull.
- **Failure (B<sub>1</sub>):** A point outside the current hull sees all edges of the convex hull.
- **Failure (B<sub>2</sub>):** A point outside the current hull sees a non-contiguous set of edges.

Failures (A<sub>1</sub>) and (A<sub>2</sub>) are equivalent to the negation of Property A. Similarly, Failures (B<sub>1</sub>) and (B<sub>2</sub>) are complete for Property B if we take (A<sub>1</sub>) into account. Are all these failures realizable? We now affirm this.

#### 4.1 Single Step Failures

We give instances violating the correctness properties of the algorithm. More precisely, we give sequences  $p_1, p_2, p_3, \dots$  of points such that the first three points form a counter-clockwise triangle (and *float\_orient* correctly discovers this) and such that the insertion of some later point leads to a violation of a correctness property (in the computations with doubles). We also discuss how we arrived at the examples. All our examples involve nearly or truly collinear points; instances without nearly collinear or truly collinear points cause no problems in view of the error analysis of the preceding section (omitted in this



**Fig. 2.** (a) The convex hull illustrating Failure (A<sub>1</sub>). The point in the lower left corner is left out of the hull. (b) schematically indicates the ridiculous situation of a point outside the current hull and seeing no edge of the hull:  $x$  lies to the left of all sides of the triangle  $(p, q, r)$ .

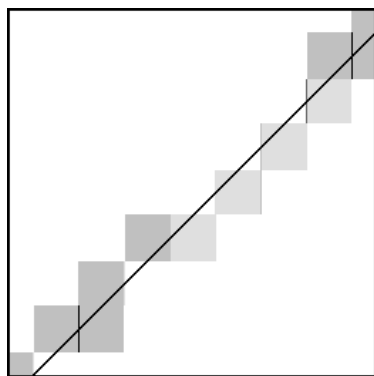
extended abstract). Does this make our examples unrealistic? We believe not. Many point sets contain nearly collinear points or truly collinear points which become nearly collinear by conversion to floating point representation.

(A<sub>1</sub>) *A point outside the current hull sees no edge of the current hull:* Consider the set of points below. Figure 2(a) shows the computed convex hull, where a point which is clearly extreme was left out of the hull.

$p_1 = (7.3000000000000194, 7.3000000000000167)$	$\text{float\_orient}(p_1, p_2, p_3) > 0,$
$p_2 = (24.000000000000068, 24.000000000000071)$	$\text{float\_orient}(p_1, p_2, p_4) > 0,$
$p_3 = (24.000000000000005, 24.000000000000053)$	$\text{float\_orient}(p_2, p_3, p_4) > 0,$
$p_4 = (0.50000000000001621, 0.50000000000001243)$	$\text{float\_orient}(p_3, p_1, p_4) > 0$ (!!).
$p_5 = (8, 4)$ $p_6 = (4, 9)$ $p_7 = (15, 27)$	
$p_8 = (26, 25)$ $p_9 = (19, 11)$	

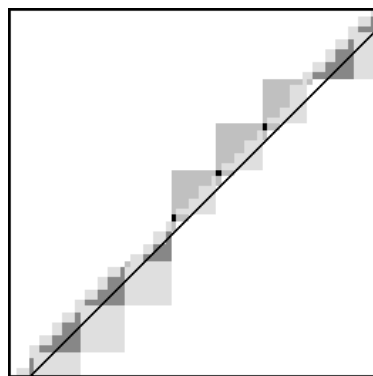
*What went wrong?* The culprits are the first four points. They lie almost on the line  $y = x$ , and  $\text{float\_orient}$  gives the results shown above. Only the last evaluation is wrong, indicated by “(!)”. Geometrically, these four evaluations say that  $p_4$  sees no edge of the triangle  $(p_1, p_2, p_3)$ . Figure 2(b) gives a schematic view of this ridiculous situation. The points  $p_5, \dots, p_9$  are then correctly identified as extreme points and are added to the hull. However, the algorithm never recovers from the error made when considering  $p_4$  and the result of the computation differs drastically from the correct hull.

We next explain how we arrived at the instance above. Intuition told us that an example (if it exists at all) would be a triangle with two almost parallel sides and with a query point near the wedge defined by the two nearly parallel edges. In view of Figure 1 such a point might be mis-classified with respect to one of the edges and hence would be unable to see any edge of the triangle. So we started with the points used in Figure 1(b), i.e.,  $p_1 \approx (17, 17)$ ,  $p_2 \approx (24, 24) \approx p_3$ , where we moved  $p_2$  slightly to the right so as to guarantee that we obtain a counter-clockwise triangle. We then probed the edges incident to  $p_1$  with points  $p_4$  in and near the wedge formed by these edges. Figure 3(a) visualizes the outcomes of the two relevant orientation tests. Each black pixel (not lying on the line indicating the nearly parallel edges of the triangle) is a candidate for Failure (A<sub>1</sub>). The



$p_1 : (17.300000000000001, 17.300000000000001)$   
 $p_2 : (24.000000000000068, 24.000000000000071)$   
 $p_3 : (24.000000000000005, 24.000000000000053)$   
 $p_4 : (0.5, 0.5)$

(a)



$(7.3000000000000194, 7.3000000000000167)$   
 $(24.000000000000068, 24.000000000000071)$   
 $(24.000000000000005, 24.000000000000053)$   
 $(0.5, 0.5)$

(b)

**Fig. 3.** The points  $(p_1, p_2, p_3)$  form a counter-clockwise triangle and we are interested in the classification of points  $(x(p_4) + xu, y(p_4) + yu)$  with respect to the edges  $(p_1, p_2)$  and  $(p_3, p_1)$  incident to  $p_1$ . The extensions of these edges are indistinguishable in the pictures and are drawn as a single black line. The black points not lying on the black diagonal do not “float-see” either one of the edges (Failure  $A_1$ ). Points collinear with one of the edges are middle grey, those collinear with both edges are light grey, those classified as seeing one but not the other edge are white, and those seeing both edges are dark grey. (a) Example starting from points in Figure 1. (b) Example that achieves “robustness” with respect to the first three points.

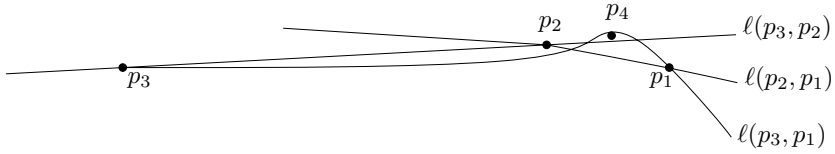
example obtained in this way was not completely satisfactory, since some orientation tests on the initial triangle  $(p_1, p_2, p_3)$  were evaluating to zero.

We perturbed the example further, aided by visualizing  $\text{float\_orient}(p_1, p_2, p_3)$ , until we found the example shown in (b); by our error analysis, this test incurs the largest error among the six possible ways of performing the orientation test for three points and is hence most likely to return the incorrect result. The final example has the nice property that all possible  $\text{float\_orient}$  tests on the first three points are correct. So this example is pretty much independent from any conceivable initialization an algorithm could use to create the first valid triangle. Figure 3(b) shows the outcomes of the two orientations tests for our final example.

*(A<sub>2</sub>) A point inside the current hull sees an edge of the current hull:* Such examples are plenty. We take any counter-clockwise triangle and chose a fourth point inside the triangle but close to one of the edges. By Figure 1 there is the chance of sign reversal. A concrete example follows:

$p_1 = (27.643564356435643, -21.881188118811881)$   
 $p_2 = (83.366336633663366, 15.544554455445542)$   
 $p_3 = (4, 4)$   
 $p_4 = (73.415841584158414, 8.8613861386138595)$

$\text{float\_orient}(p_1, p_2, p_3) > 0$   
 $\text{float\_orient}(p_1, p_2, p_4) < 0$  (!!)  
 $\text{float\_orient}(p_2, p_3, p_4) > 0$   
 $\text{float\_orient}(p_3, p_1, p_4) > 0$



**Fig. 4.** Schematics: The point  $p_4$  sees all edges of the triangle  $(p_1, p_2, p_3)$ .

The convex hull is correctly initialized to  $(p_1, p_2, p_3)$ . The point  $p_4$  is inside the current convex hull, but the algorithm incorrectly believes that  $p_4$  can see the edge  $(p_1, p_2)$  and hence changes the hull to  $(p_1, p_4, p_2, p_3)$ , a slightly non-convex polygon.

*(B<sub>1</sub>) A point outside the current hull sees all edges of the convex hull:* Intuition told us that an example (if it exists) would consist of a triangle with one angle close to  $\pi$  and hence three almost parallel sides. Where should one place the query point? We first placed it in the extension of the three parallel sides and quite a distance away from the triangle. This did not work. The choice which worked is to place the point near one of the sides so that it could see two of the sides and “float-see” the third. Figure 4 illustrates this choice. A concrete example follows:

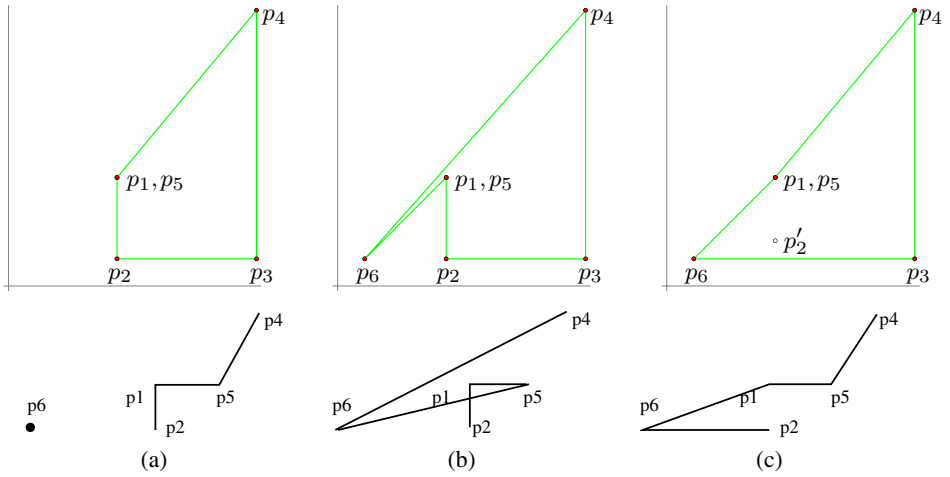
$p_1 = (200, 49.200000000000003)$	$\text{float\_orient}(p_1, p_2, p_3) > 0$
$p_2 = (100, 49.600000000000001)$	$\text{float\_orient}(p_1, p_2, p_4) < 0$
$p_3 = (-233.33333333333334, 50.93333333333333)$	$\text{float\_orient}(p_2, p_3, p_4) < 0$
$p_4 = (166.66666666666669, 49.333333333333336)$	$\text{float\_orient}(p_3, p_1, p_4) < 0 (!!)$

The first three points form a counter-clockwise oriented triangle and according to *float\_orient*, the algorithm believes that  $p_4$  can see all edges of the triangle. What will our algorithm do? It depends on the implementation details. If the algorithm first searches for an invisible edge, it will search forever and never terminate. If it deletes points on-line from  $L$  it will crash or compute nonsense depending on the details of the implementation.

*(B<sub>2</sub>) A point outside the current hull sees a non-contiguous set of edges:* Consider the following points:

$p_1 = (0.50000000000001243, 0.50000000000000189)$	$\text{float\_orient}(p_1, p_4, p_5) < 0 (!!)$
$p_2 = (0.50000000000001243, 0.500000000000000333)$	$\text{float\_orient}(p_4, p_3, p_5) > 0$
$p_3 = (24.000000000000005, 24.000000000000053)$	$\text{float\_orient}(p_3, p_2, p_5) < 0$
$p_4 = (24.000000000000068, 24.000000000000071)$	$\text{float\_orient}(p_2, p_1, p_5) > 0$
$p_5 = (17.300000000000001, 17.300000000000001)$	

Inserting the first four points results in the convex quadrilateral  $(p_1, p_4, p_3, p_2)$ ; this is correct. The last point  $p_5$  sees only the edge  $(p_3, p_2)$  and none of the other three. However, *float\_orient* makes  $p_5$  see also the edge  $(p_1, p_4)$ . The subsequences of visible and invisible edges are not contiguous. Since the falsely classified edge  $(p_1, p_4)$  comes first, our algorithm inserts  $p_5$  at this edge, removes no other vertex, and returns a polygon that has self-intersections and is not simple.



**Fig. 5.** (a) The hull constructed after processing points  $p_1$  to  $p_5$ . Points  $p_1$  and  $p_5$  lie close to each other and are indistinguishable in the upper figure. The schematic sketch below shows that we have a concave corner at  $p_5$ . The point  $p_6$  sees the edges  $(p_1, p_2)$  and  $(p_4, p_5)$ , but does **not** see the edge  $(p_5, p_1)$ . One of the former edges will be chosen by the algorithm as the chain of edges visible from  $p_6$ . Depending on the choice, we obtain the hulls shown in (b) or (c). In (b),  $(p_4, p_5)$  is found as the visible edge, and in (c),  $(p_1, p_2)$  is found. We refer the reader to the text for further explanations. The figures show the coordinate axes for orientation.

## 4.2 Global Effects

By now, we have seen examples which cover the negation space of the correctness properties on the incremental algorithm and we have seen the effect of an incorrect orientation test for a single update step. We next study global effects. *The goal is to refute the myth that the algorithm will always compute an approximation of the true convex hull.*

*The algorithm computes a convex polygon, but misses some of the extreme points:* We have already seen such an example in Failure (A<sub>1</sub>). We can modify this example so that the ratio of the areas of the true hull and the computed hull becomes arbitrarily large. We do as in Failure (A<sub>1</sub>), but move the fourth point to infinity. The true convex hull has four extreme points. The algorithm misses  $q_4$ .

$$\begin{aligned}
 q_1 &= (0.100000000000000001, 0.100000000000000001) & \text{float\_orient}(q_1, q_2, q_3) < 0 \\
 q_2 &= (0.200000000000000001, 0.200000000000000004) & \text{float\_orient}(q_1, q_2, q_4) = 0 \text{ (!!)} \\
 q_3 &= (0.79999999999999993, 0.800000000000000004) & \text{float\_orient}(q_2, q_3, q_4) = 0 \text{ (!!)} \\
 q_4 &= (1.267650600228229 \cdot 10^{30}, 1.2676506002282291 \cdot 10^{30}) & \text{float\_orient}(q_3, q_1, q_4) > 0
 \end{aligned}$$

*The algorithm crashes or does not terminate:* See Failure (B<sub>1</sub>).



*The algorithm computes a non-convex polygon:* We have already given such an example in Failure (A<sub>2</sub>). However, this failure is not visible to the naked eye. We next give examples where non-convexity is visible to the naked eye. We consider the points:

$$\begin{aligned} p_1 &= (24.00000000000005, 24.000000000000053) & p_2 &= (24.0, 6.0) \\ p_3 &= (54.85, 6.0) & p_4 &= (54.850000000000357, 61.000000000000121) \\ p_5 &= (24.000000000000068, 24.000000000000071) & p_6 &= (6, 6). \end{aligned}$$

After the insertion of  $p_1$  to  $p_4$ , we have the convex hull  $(p_1, p_2, p_3, p_4)$ . This is correct. Point  $p_5$  lies inside the convex hull of the first four points; but  $\text{float\_orient}(p_4, p_1, p_5) < 0$ . Thus  $p_5$  is inserted between  $p_4$  and  $p_1$ , and we obtain  $(p_1, p_2, p_3, p_4, p_5)$ . However, this error is not visible yet to the eye, see Figure 5(a).

The point  $p_6$  sees the edges  $(p_4, p_5)$  and  $(p_1, p_2)$ , but does not see the edge  $(p_5, p_1)$ . All of this is correctly determined by  $\text{float\_orient}$ . Consider now the insertion process for point  $p_6$ . Depending on where we start the search for a visible edge, we will either find the edge  $(p_4, p_5)$  or the edge  $(p_1, p_2)$ . In the former case, we insert  $p_6$  between  $p_4$  and  $p_5$  and obtain the polygon shown in (b). It is visibly non-convex and has a self-intersection. In the latter case, we insert  $p_6$  between  $p_1$  and  $p_2$  and obtain the polygon shown in (c). It is visibly non-convex.

Of course, in a deterministic implementation, we will see only one of the errors, namely (b). This is because in our example, the search for a visible edge starts at edge  $(p_2, p_3)$ . In order to produce (c) with the implementation we replace the point  $p_2$  by the point  $p'_2 = (24.0, 10.0)$ . Then  $p_6$  sees  $(p'_2, p_3)$  and identifies  $(p_1, p'_2, p_3)$  as the chain of visible edges and hence constructs (c).

## 5 Conclusion

We provided instances which cause the floating point implementation of a simple convex hull algorithm to fail in many different ways. We showed how to construct such instances semi-systematically. We hope that our paper and its companion web page will be useful for classroom use and that it will alert students and researchers to the intricacies of implementing geometric algorithms.

What can be done to overcome the robust problem of floating point arithmetic? There are essentially three approaches: (1) make sure that the implementation of the orientation predicate always returns the correct result or (2) change the algorithm so that it can cope with the floating point implementation of the orientation predicate and still computes something meaningful or (3) perturb the input so that the floating point implementation is guaranteed to produce the correct result on the perturbed input. The first approach is the most general and is known under the the exact geometric computation (EGC) paradigm and was first used by Jünger, Reinelt and Zepf [JRZ91] and Karasick, Lieber and Nackmann [KLN91]. For detailed discussions, we refer the reader to [Yap04] and Chapter 9 of [MN99]. The EGC approach has been adopted for the software libraries LEDA and CGAL and other successful implementations. The second and third approaches have been successfully applied to a number of geometric problems, see for example [Mil89, FM91, LM90, DSB92, SIII00, HS98]. But in the second approach the interpretation of “meaningful” is a crucial and difficult problem. The third approach

should in principle be applicable to all problems with purely numerical input. There are also suggested approaches, e.g., epsilon-tweaking, which do not work. Epsilon-tweaking simply activates rounding to zero, both for correct and mis-classified orientations. E.g., it is now more likely for points outside the current hull not to see any edges because of enforced collinearity.

## References

- [And79] A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9:216–219, 1979.
- [CS89] K.L. Clarkson and P.W. Shor. Applications of random sampling in computational geometry, II. *Journal of Discrete and Computational Geometry*, 4:387–421, 1989.
- [DSB92] T. K. Dey, K. Sugihara, and C. L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Comput. Aided Geom. Design*, 9:457–470, 1992.
- [FM91] S. Fortune and V.J. Milenkovic. Numerical stability of algorithms for line arrangements. In *SoCG'91*, pages 334–341. ACM Press, 1991.
- [For85] A. R. Forrest. Computational geometry in practice. In R. A. Earnshaw, editor, *Fundamental Algorithms for Computer Graphics*, volume F17 of NATO ASI, pages 707–724. Springer-Verlag, 1985.
- [Gol90] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1990.
- [Gra72] R.L. Graham. An efficient algorithm for determining the convex hulls of a finite point set. *Information Processing Letters*, 1:132–133, 1972.
- [HS98] D. Halperin and C. R. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *Comp. Geom.: Theory and Applications*, 10, 1998.
- [JRZ91] M. Jünger, G. Reinelt, and D. Zepf. Computing correct Delaunay triangulations. *Computing*, 47:43–49, 1991.
- [KLN91] M. Karasick, D. Lieber, and L.R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, January 1991.
- [LM90] Z. Li and V.J. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. In *SoCG'90*, pages 235–243. ACM Press, 1990.
- [Mil89] V.J. Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. In *SoCG'89*, pages 197–207. ACM Press, 1989.
- [MN99] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. 1018 pages.
- [She97] J.R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.
- [SIII00] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-oriented implementation - an approach to robust geometric algorithms. *Algorithmica*, 27(1):5–20, 2000.
- [Yap04] C. K. Yap. Robust geometric computation. In J.E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41. CRC Press LLC, Boca Raton, FL, 2nd edition, 2004.

# Stable Minimum Storage Merging by Symmetric Comparisons

Pok-Son Kim<sup>1\*</sup> and Arne Kutzner<sup>2</sup>

<sup>1</sup> Kookmin University, Department of Mathematics, Seoul 136-702, Rep. of Korea  
pskim@kookmin.ac.kr

<sup>2</sup> Seokyeong University, Department of E-Business, Seoul 136-704, Rep. of Korea  
kutzner@skuniv.ac.kr

**Abstract.** We introduce a new stable minimum storage algorithm for merging that needs  $O(m \log(\frac{n}{m} + 1))$  element comparisons, where  $m$  and  $n$  are the sizes of the input sequences with  $m \leq n$ . According to the lower bound for merging, our algorithm is asymptotically optimal regarding the number of comparisons.

The presented algorithm rearranges the elements to be merged by rotations, where the areas to be rotated are determined by a simple principle of symmetric comparisons. This style of minimum storage merging is novel and looks promising.

Our algorithm has a short and transparent definition. Experimental work has shown that it is very efficient and so might be of high practical interest.

## 1 Introduction

Merging denotes the operation of rearranging the elements of two adjacent sorted sequences of sizes  $m$  and  $n$ , so that the result forms one sorted sequence of  $m + n$  elements. An algorithm merges two adjacent sequences with *minimum storage* [10] when it needs  $O(\log^2(m + n))$  bits additional space at most. This form of merging represents a weakened form of *in-place* merging and allows the usage of a stack that is logarithmically bounded in  $m + n$ . Minimum storage merging is sometimes also referred to as *in situ* merging. A merging algorithm is regarded as *stable*, if it preserves the initial ordering of elements with equal value.

Some lower bounds for merging have been proven so far. The lower bound for the number of assignments is  $m + n$ , because every element may change its position in the sorted result. The lower bound for the number of comparisons is  $O(m \log \frac{n}{m})$  for  $m \leq n$ . This can be proven by a combinatorial inspection combined with an argumentation using decision trees. An accurate presentation of these bounds is given by Knuth [10].

---

\* This work was supported by the Kookmin University research grant in 2004.

The simple standard merge algorithm is rather inefficient, because it uses linear extra space and always needs a linear number of comparisons. In particular the need of extra space motivated the search for efficient in-place merging algorithms. The first publication presenting an in-place merging algorithm was due to Kronrod [11] in 1969. Kronrod's unstable merge algorithm is based on a partition merge strategy and uses an internal buffer as central component. This strategy has been refined and improved in numerous subsequent publications. A selection of these publications is [7,8,9,3,12,15,17,16]; an accurate description of the history and evolution of Kronrod-related algorithms can be found in [3]. The more recent Kronrod-related publications, like [7] and [3], present quite complex algorithms. The correctness of these algorithms is by no means immediately clear.

A minimum storage merging algorithm that isn't Kronrod-related was proposed by Dudzinski and Dydek [5] in 1981. They presented a divide and conquer algorithm that is asymptotically optimal regarding the number of comparisons but nonlinear regarding the number of assignments. This algorithm was chosen as basis of the implementation of the `merge_without_buffer` function in the C++ Standard Template Libraries[2], possibly due to its transparent nature and short definition.

A further method was proposed by Ellis and Markov in [6], where they introduce a shuffle-based in situ merging strategy. But their algorithm needs  $((n + m) \log(n + m))$  assignments and  $((n + m) \log(n + m))$  comparisons. So, despite some practical value, the algorithm isn't optimal from the theoretical point of view.

We present a new stable minimum storage merging algorithm performing  $O(m \log \frac{n}{m})$  comparisons and  $O((m + n) \log m)$  assignments for two sequences of size  $m$  and  $n$  ( $m \leq n$ ). Our algorithm is based on a simple strategy of symmetric comparisons, which will be explained in detail by an example. We report about some benchmarking showing that the proposed algorithm is fast and efficient compared to other minimum storage merging algorithms as well as the standard algorithm. We will finish with a conclusion, where we give a proposal for further research.

## 2 The SYMMERGE Algorithm

We start with a brief introduction of our approach to merging. Let us assume that we have to merge the two sequences  $u = (0, 2, 5, 9)$  and  $v = (1, 4, 7, 8)$ . When we compare the input with the sorted result, we can see that in the result the last two elements of  $u$  occur on positions belonging to  $v$ , and the first two elements of  $v$  appear on positions belonging to  $u$  (see Fig. 1 a)). So, 2 elements were exchanged between  $u$  and  $v$ . The kernel of our algorithm is to compute this number of side-changing elements efficiently and then to exchange such a number of elements. More accurately, if we have to exchange  $n$  ( $n \geq 0$ ) elements between sequences  $u$  and  $v$ , we move the  $n$  greatest elements from  $u$  to  $v$  and

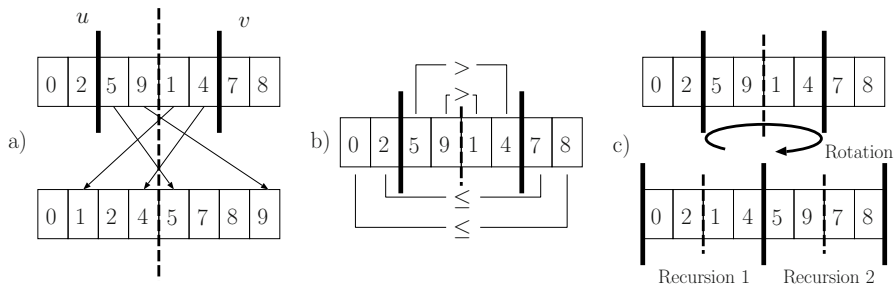


Fig. 1. SYMMERGE example

the  $n$  smallest elements from  $v$  to  $u$ , where the exchange of elements is realized by a rotation. Then by recursive application of this technique to the arising subsequences we get a sorted result. Fig. 1 illustrates this approach for our above example.

We will now focus on the process of determining the number of elements to be exchanged. This number can be determined by a process of symmetrical comparisons of elements that happens according to the following principle:

We start at the leftmost element in  $u$  and at the rightmost element in  $v$  and compare the elements at these positions. We continue doing so by symmetrically comparing element-pairs from the outsides to the middle. Fig. 1 b) shows the resulting pattern of mutual comparisons for our example. There can occur at most one position, where the relation between the compared elements alters from 'not greater' to 'greater'. In Figure 1 b) two thick lines mark this position. These thick lines determine the number of side-changing elements as well as the bounds for the rotation mentioned above.

Due to this technique of symmetric comparisons we will call our algorithm SYMMERGE. Please note, if the bounds are on the leftmost and rightmost position, this means all elements of  $u$  are greater than all elements of  $v$ , we exchange  $u$  and  $v$  and get immediately a sorted result. Conversely, if both bounds meet in the middle we terminate immediately, because  $uv$  is then already sorted. So our algorithm can take advantage of the sortedness of the input sequences.

So far we introduced the computation of the number of side-changing elements as linear process of symmetric comparisons. But this computation may also happen in the style of a binary search. Then only  $\lfloor \log(\min(|u|, |v|)) \rfloor + 1$  comparisons are necessary to compute the number of side-changing elements.

### 2.1 Formal Definition

Let  $u$  and  $v$  be two adjacent ascending sorted sequences. We define  $u \leq v$  ( $u < v$ ) iff.  $x \leq y$  ( $x < y$ ) for all elements  $x \in u$  and for all elements  $y \in v$ .

We merge  $u$  and  $v$  as follows:

If  $|u| \leq |v|$ , then

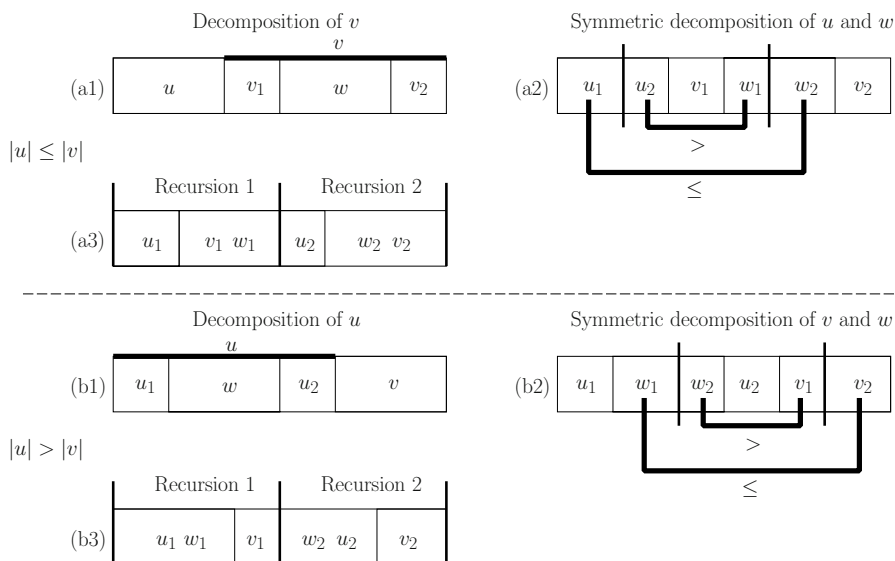
- (a1) we decompose  $v$  into  $v_1 w v_2$  such that  $|w| = |u|$  and either  $|v_2| = |v_1|$  or  $|v_2| = |v_1| + 1$ .
- (a2) we decompose  $u$  into  $u_1 u_2$  ( $|u_1| \geq 0, |u_2| \geq 0$ ) and  $w$  into  $w_1 w_2$  ( $|w_1| \geq 0, |w_2| \geq 0$ ) such that  $|u_1| = |w_2|, |u_2| = |w_1|$  and  $u_1 \leq w_2, w_2 > w_1$ .
- (a3) we recursively merge  $u_1$  with  $v_1 w_1$  as well as  $u_2$  with  $w_2 v_2$ . Let  $u'$  and  $v'$  be the resulting sequences, respectively.

else

- (b1) we decompose  $u$  into  $u_1 w u_2$  such that  $|w| = |v|$  and either  $|u_2| = |u_1|$  or  $|u_2| = |u_1| + 1$ .
- (b2) we decompose  $v$  into  $v_1 v_2$  ( $|v_1| \geq 0, |v_2| \geq 0$ ) and  $w$  into  $w_1 w_2$  ( $|w_1| \geq 0, |w_2| \geq 0$ ) such that  $|v_1| = |w_2|, |v_2| = |w_1|$  and  $w_1 \leq v_2, w_2 > v_1$ .
- (b3) we recursively merge  $u_1 w_1$  with  $v_1$  as well as  $w_2 u_2$  with  $v_2$ . Let  $u'$  and  $v'$  be the resulting sequences, respectively.

$u'v'$  then contains all elements of  $u$  and  $v$  in sorted order.

Fig. 2 contains an accompanying graphical description of the process described above. The steps (a1) and (b1) manage the situation of input sequences of different length by cutting a subsection  $w$  in the middle of the longer sequence as “active area”. This active area has the same size as the shorter of either input sequences. The decomposition formulated by the steps (a2) and (b2) can



**Fig. 2.** Illustration of SYMMERGE

**Algorithm 1** SYMMERGE algorithm

---

```

SYMMERGE ( $A, first1, first2, last$ )
  if  $first1 < first2$  and  $first2 < last$  then
     $m \leftarrow (first1 + last) / 2$ 
     $n \leftarrow m + first2$ 
    if  $first2 \nmid m$  then
       $start \leftarrow \text{BSEARCH}(A, n - last, m, n - 1)$ 
    else
       $start \leftarrow \text{BSEARCH}(A, first1, first2, n - 1)$ 
     $end \leftarrow n - start$ 
    ROTATE ( $A, start, first2, end$ )
    SYMMERGE ( $A, first1, start, m$ )
    SYMMERGE ( $A, m, end, last$ )

BSEARCH ( $A, l, r, p$ )
  while  $l \nmid r$ 
     $m \leftarrow (l + r) / 2$ 
    if  $A[m] \leq A[p - m]$ 
      then  $l \leftarrow m + 1$ ;
    else  $r \leftarrow m$ ;
  return  $l$ 

```

---

be achieved efficiently by applying the principle of the symmetric comparisons between the shorter sequence  $u$  (or  $v$ ) and the active area  $w$ . After the decomposition step (a2) (or (b2)), the subsequence  $u_2v_1w_1$  (or  $w_2u_2v_1$ ) is rotated so that we get the subsequences  $u_1v_1w_1$  and  $u_2w_2v_2$  ( $u_1w_1v_1$  and  $w_2u_2v_2$ ). The treatment of pairs of equal elements as part of the “outer blocks” ( $u_1, w_2$  in (a2) and  $w_1, v_2$  in (b2)) avoids the exchange of equal elements and so any reordering of these.

**Corollary 1.** SYMMERGE is stable.

Algorithm 1 gives an implementation of the SYMMERGE algorithm in Pseudocode. The Pseudocode conventions are taken from [4].

### 3 Worst Case Complexity

We will now investigate the worst case complexity of SYMMERGE regarding the number of comparisons and assignments.

Unless stated otherwise, let us denote  $m = |u|$ ,  $n = |v|$ ,  $m \leq n$ ,  $k = \lfloor \log m \rfloor$  and let  $m_j^i$  and  $n_j^i$  denote the minimum and maximum of lengths of sequences merging on the  $i$ th recursion group for  $i = 0, 1, \dots, k$  and  $j = 1, 2, 3, \dots, 2^i$  (initially  $m_1^0 = m$  and  $n_1^0 = n$ ). A recursion group consists of one or several

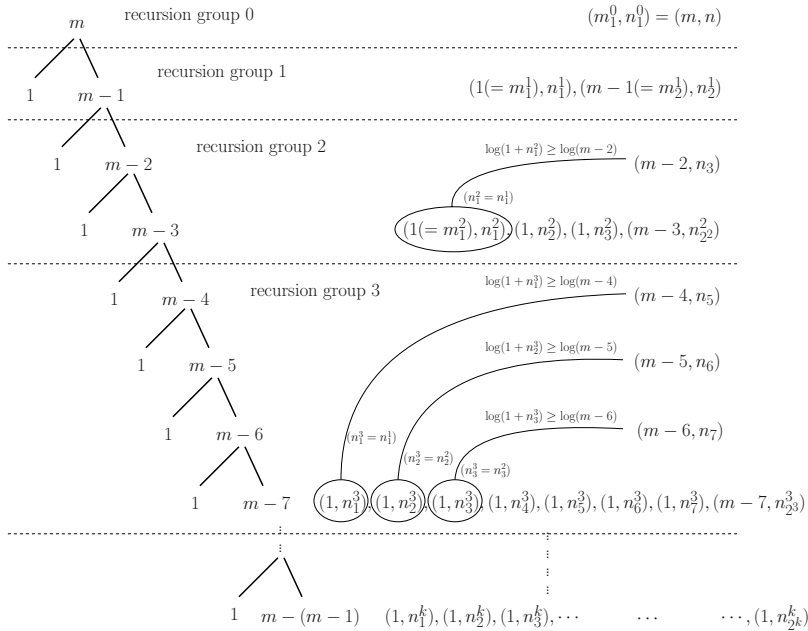


Fig. 3. Maximum spanning case

recursion levels and comprises  $2^i (i = 0, 1, \dots, k)$  subsequence mergings at most (see figure 3). In the special case where each subsequence merging always triggers two nonempty recursive calls - in this case the recursion depth becomes exactly  $k = \lfloor \log m \rfloor$  -, recursion groups and recursion levels are identical, but in general for the recursion depth  $dp$  it holds  $k = \lfloor \log m \rfloor \leq dp \leq m$ . Further, for each recursion group  $i = 0, 1, \dots, k$ , it holds  $\sum_{j=1}^{2^i} (m_j^i + n_j^i) = m + n$ .

**Lemma 1.** ([5] Lemma 3.1) If  $k = \sum_{j=1}^{2^i} k_j$  for any  $k_j > 0$  and integer  $i \geq 0$ , then  $\sum_{j=1}^{2^i} \log k_j \leq 2^i \log(k/2^i)$ .

**Theorem 1.** The SYMMERGE algorithm needs  $O(m \log(n/m + 1))$  comparisons.

*Proof.* The number of comparisons for the binary search for the recursion group 0 is equal to  $\lfloor \log m \rfloor + 1 \leq \lfloor \log m + n \rfloor + 1$ . For the recursion group 1 we need at most  $\log(m_1^1 + n_1^1) + 1 + \log(m_2^1 + n_2^1) + 1$  comparisons, and so on. For the recursion group  $i$  we need at most  $\sum_{j=1}^{2^i} \log(m_j^i + n_j^i) + 2^i$  comparisons. Since  $\sum_{j=1}^{2^i} (m_j^i + n_j^i) = m + n$ , it holds  $\sum_{j=1}^{2^i} \log(m_j^i + n_j^i) + 2^i \leq 2^i \log((m+n)/2^i) + 2^i$  by Lemma 1. So the overall number of comparisons for all  $k + 1$  recursion groups is not greater than  $\sum_{i=0}^k (2^i + 2^i \log((m+n)/2^i)) = 2^{k+1} - 1 + (2^{k+1} - 1) \log(m+n) - \sum_{i=0}^k i 2^i$ . Since



$\sum_{i=0}^k i2^i = (k-1)2^{k+1} + 2$ , algorithm SYMMERGE needs at most  $2^{k+1} - 1 + (2^{k+1} - 1) \log(m+n) - (k-1)2^{k+1} - 2 \leq 2^{k+1} \log(m+n) - k2^{k+1} + 2^{k+2} - \log(m+n) - 3 \leq 2m(\log \frac{m+n}{m} + 2) - \log(m+n) - 3 = O(m \log(\frac{n}{m} + 1))$  comparisons.  $\square$

**Theorem 2.** *The recursion-depth of SYMMERGE is bounded by  $\lceil \log(m+n) \rceil$ .*

*Proof.* The decomposition steps (1a) and (1b) satisfy the property  $\max\{|u_1|, |u_2|, |v_1|, |v_2|, |w_1|, |w_2|\} \leq (|u| + |v|)/2$ . So, on recursion level  $\lceil \log(m+n) \rceil$  all remaining unmerged subsequences are of length 1.  $\square$

**Corollary 2.** *SYMMERGE is a minimum storage algorithm.*

In [5] Dudzinski and Dydek presented an optimal in-place rotation (sequence exchange) algorithm, that needs  $m + n + \gcd(m, n)$  element assignments for exchanging two sequences of lengths  $m$  and  $n$ .

**Theorem 3.** *If we take the rotation algorithm given in [5], then SYMMERGE requires  $O((m+n) \log m)$  element assignments.*

*Proof.* Inside each recursion group  $i = 0, 1, \dots, k$  disjoint parts of  $u$  are merged with disjoint parts of  $v$ . Hence each recursion group  $i$  comprises at most  $\sum_{j=1}^i ((m_j^i + n_j^i) + \gcd(m_j^i, n_j^i)) \leq m + n + \sum_{j=1}^{2^i} m_j^i = 2m + n$  assignments resulted from rotations. So the overall number of assignments for all  $k$  recursion groups is less than  $(2m+n)(k+1) = (2m+n) \log m + 2m+n = O((m+n) \log m)$ .  $\square$

## 4 Practical Results

We did some experimental work with the unfolded version of the SYMMERGE algorithm (see Sect. 4.1) and compared it with the implementations of three other merging algorithms. As first competitor we chose the `merge_without_buffer` function contained in the C++ Standard Template Libraries (STL) [2]. This function implements a modified version of the RECMERGE algorithm devised by Dudzinski and Dydek [5]. The STL-algorithm operates in a mirrored style compared to RECMERGE and doesn't release one element in each recursion step. Fig. 4 gives a graphical description of the differences. We preferred the STL-variant because of its significance in practice.

The second competitor was taken from [14], where a simplified implementation of the in-place algorithm from Mannila & Ukkonen [12] is given. Unlike the original algorithm the simplified version relies on a small external buffer whose length is restricted by the square root of the input size.

As third competitor we took the classical standard algorithm. We chose randomly generated sequences of integers as input. The results of our evaluation

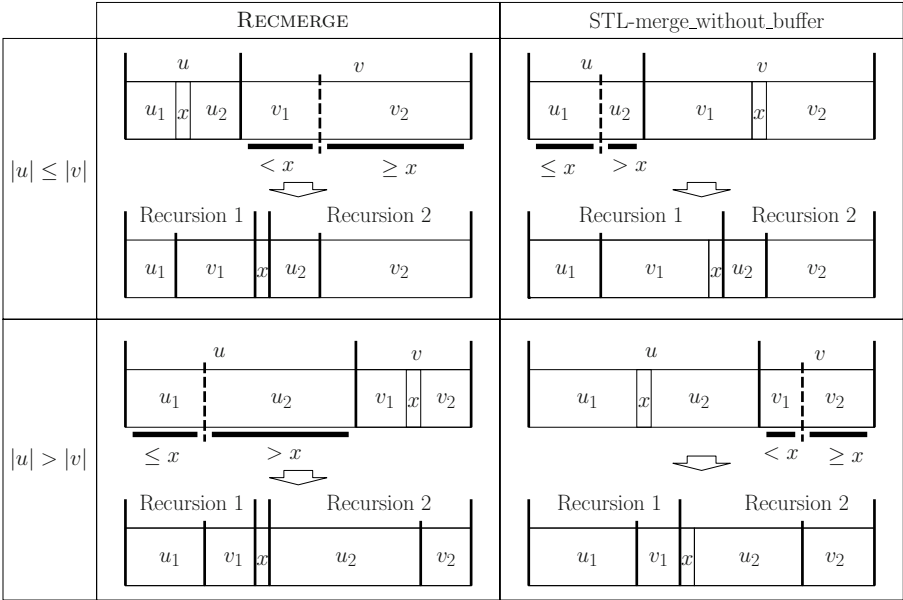


Fig. 4. RECMEERGE versus STL-merge\_without\_buffer

Table 1. Practical comparison of various merging algorithms

<i>n</i>	<i>m</i>	SYMMERGE		STL <sub>modified</sub>		STL <sub>orig</sub>	Mannila & Ukkonen		Standard	
		#comp	<i>t<sub>e</sub></i>	#comp	<i>t<sub>e</sub></i>		#comp	<i>t<sub>e</sub></i>	#comp	<i>t<sub>e</sub></i>
2 <sup>21</sup>	2 <sup>21</sup>	5217738	1831	6020121	2263	3408	7464568	1483	4194177	743
2 <sup>21</sup>	2 <sup>18</sup>	1348902	635	1541524	751	1398	5615225	1116	2359159	415
2 <sup>21</sup>	2 <sup>15</sup>	264279	298	295063	334	815	5301311	1049	2129765	372
2 <sup>21</sup>	2 <sup>12</sup>	45227	211	49272	231	607	4660337	942	2100744	373
2 <sup>23</sup>	2 <sup>9</sup>	8198	680	8711	777	2735	14909001	3101	8374604	1503
2 <sup>23</sup>	2 <sup>6</sup>	1212	579	1279	673	1254	14297482	2996	8292870	1496
2 <sup>23</sup>	2 <sup>3</sup>	170	465	179	548	196	14202829	2967	7544491	1402
2 <sup>23</sup>	2 <sup>0</sup>	23	191	24	222	31	14188252	2959	4040739	832

*t<sub>e</sub>* : Execution time in ms, #comp : Number of comp., *m*, *n* : Lengths of inp. seq.

are contained in Table 1, each entry shows a mean value of 30 runs with different data. We took a state of the art hardware platform with 2.4 Ghz processor speed and 512MB main memory; all coding was done in the C-programming language. Each comparison was accompanied by a slight artificial delay in order to strengthen the impact of comparisons on the execution time. It can be read from the table that the STL-algorithm seems to be less efficient than SYMMERGE. However both algorithm show an accurate “ $O(m \log(n/m))$ ”-behavior”.

SYMMERGE and RECMERGE rely on rotations as encapsulated operations for element reordering. There are several algorithms that achieve rotations, three of them are presented and evaluated by Bentley in [1]. The evaluation by Bentley showed, that a rotation algorithm proposed by Dudzinski and Dydek in [5] is rather slow compared to its alternatives, although it is optimal with respect to the number of assignments. Nevertheless this algorithm was chosen in the STL for the implementation of rotations. We exchanged the rotation algorithm of the `merge_without_buffer`-function (STL) by one of its faster alternatives and compared the modified function with its original. The result of this comparisons is given in the columns  $STL_{modified}$  and  $STL_{orig}$  of Table 1. From this comparison it can be read that the chosen rotation algorithm is one of the driving factors regarding the performance of RECMERGE, and so of SYMMERGE.

#### 4.1 Optimisations on Pseudocode-Level

We would like to remark two optimisations of SYMMERGE on Pseudocode-Level that can be applied for decreasing the execution time without changing the number of comparisons and element assignments carried out. First of all, “needless recursive calls”, i.e. recursive calls with  $m = 0$  or  $n = 0$ , can be avoided by unfolding the algorithm. For the unfolded version the caller has to ensure that SYMMERGE is called with  $m \geq 1$  and  $n \geq 1$ . A second optimisation is the treatment of the case  $m = 1$  and  $n \geq 1$  as loop-driven direct binary insertion. This avoids  $\lfloor \log n + 1 \rfloor$  recursive calls of the algorithm in this special case. The impact of both optimizations on the execution time depends on the sequences to be merged. In our environment we could observe a time-reduction up to 25%.

## 5 Conclusion

We presented an efficient minimum storage merging algorithm called SYMMERGE. Our algorithm uses a novel technique for merging that relies on symmetric comparisons as central operation. We could prove that our algorithm is asymptotically optimal regarding the number of necessary comparisons. Practical evaluation could show that it is fast and efficient. So, SYMMERGE is not only of theoretical but also of practical interest.

Finally let us note that our algorithm, unlike the standard algorithm, can take advantage of the sortedness of the input sequences. If the overall input sequence is in sorted order, i.e.  $u \leq v$  for two sequences  $u$  and  $v$  of sizes  $m$  and  $n$ , SYMMERGE needs only  $O(\log(m + n))$  comparisons and hence becomes sub-linear. This in turn reflects to a SYMMERGE based Merge-sort, which speeds up for pre-sorted sequences as well. Mehlhorn showed in [13] that sequences of size  $n$  with  $O(n)$  inversions, i.e. with  $O(n)$  pairs of elements that are not in sorted order, can be sorted in time  $O(n)$ . How many comparisons does a SYMMERGE based Merge-sort need depending on the number of inversions? We would like to leave this question to the further research.

**Acknowledgment.** We are grateful to a referee whose careful reading and valuable remarks helped to improve the contents and presentation of the paper.

## References

1. J. Bentley. *Programming Pearls*. Addison-Wesley, Inc, 2nd edition, 2000.
2. C++ Standard Template Library. <http://www.sgi.com/tech/stl>.
3. J. Chen. Optimizing stable in-place merging. *Theoretical Computer Science*, 302(1/3):191–210, 2003.
4. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.
5. K. Dudzinski and A. Dydek. On a stable storage merging algorithm. *Information Processing Letters*, 12(1):5–8, February 1981.
6. John Ellis and Minko Markov. In situ, stable merging by way of the perfect shuffle. *The Computer Journal*, 43(1):40–53, 2000.
7. V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, 237(1/2):159–181, 2000.
8. B-C. Huang and M.A. Langston. Practical in-place merging. *Communications of the ACM*, 31:348–352, 1988.
9. B-C. Huang and M.A. Langston. Fast stable merging and sorting in constant extra space. *The Computer Journal*, 35(6):643–650, 1992.
10. D. E. Knuth. *The Art of Computer Programming*, volume Vol. 3: Sorting and Searching. Addison-Wesley, 1973.
11. M. A. Kronrod. An optimal ordering algorithm without a field operation. *Dokladi Akad. Nauk SSSR*, 186:1256–1258, 1969.
12. H. Mannila and Esko Ukkonen. A simple linear-time algorithm for in situ merging. *Information Processing Letters*, 18:203–208, 1984.
13. K. Mehlhorn. Sorting presorted files. In *Proc. 4th GI Conf. on Theoretical Comp. Science (Aachen) - Lect. Notes in Comp. Science 67*, pages 199–212. Springer, 1979.
14. K. Møllerhøj and C.U. Søttrup. Undersøgelse og implementation af effektiv inplace merge. Technical report, CPH STL Reports 2002-06, Department of Computer Science, University of Copenhagen, Denmark, June 2002.
15. L. T. Pardo. Stable sorting and merging with optimal space and time bounds. *SIAM Journal on Computing*, 6(2):351–372, June 1977.
16. J. Salowe and W. Steiger. Simplified stable merging tasks. *Journal of Algorithms*, 8:557–571, 1987.
17. A. Symvonis. Optimal stable merging. *Computer Journal*, 38:681–690, 1995.

# On Rectangular Cartograms

Marc van Kreveld<sup>1</sup> and Bettina Speckmann<sup>2</sup>

<sup>1</sup> Inst. for Information & Computing Sciences, Utrecht University, [marc@cs.uu.nl](mailto:marc@cs.uu.nl)

<sup>2</sup> Dep. of Mathematics & Computer Science, TU Eindhoven, [speckman@win.tue.nl](mailto:speckman@win.tue.nl)

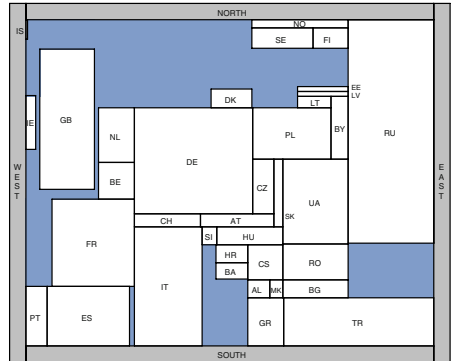
**Abstract.** A rectangular cartogram is a type of map where every region is a rectangle. The size of the rectangles is chosen such that their areas represent a geographic variable (e.g., population). Good rectangular cartograms are hard to generate: The area specifications for each rectangle may make it impossible to realize correct adjacencies between the regions and so hamper the intuitive understanding of the map.

Here we present the first algorithms for rectangular cartogram construction. Our algorithms depend on a precise formalization of region adjacencies and are building upon existing VLSI layout algorithms. Furthermore, we characterize a non-trivial class of rectangular subdivisions for which exact cartograms can be efficiently computed. An implementation of our algorithms and various tests show that in practice, visually pleasing rectangular cartograms with small cartographic error can be effectively generated.

## 1 Introduction

**Cartograms.** Cartograms, which are also referred to as *value-by-area maps*, are a useful and intuitive tool to visualize statistical data about a set of regions like countries, states or provinces. The size of a region in a cartogram corresponds to a particular geographic variable [3,10]. Since the sizes of the regions are not their true sizes they generally cannot keep both their shape and their adjacencies. A good cartogram, however, preserves the recognizability in some way.

Globally speaking, there are three types of cartogram. The standard type (the *contiguous area cartogram*) has deformed regions so that the desired sizes can be obtained and the adjacencies kept. Algorithms for such cartograms are described in [4,5,7,15]. The second type of cartogram is the non-contiguous area cartogram [11]. The regions have the true shape, but are scaled down and generally do not touch anymore. The third type of cartogram is the rectangular



**Fig. 1.** The population of Europe (country codes according to ISO 3611).

cartogram introduced by Raisz in 1934 [12] where each region is represented by a rectangle. This has the great advantage that the sizes (area) of the regions can be estimated much better than with the first two types. However, the rectangular shape is less recognizable and it imposes limitations on the possible layout.

**Quality criteria.** Whether a rectangular cartogram is good is determined by several factors. One of these is the *cartographic error* [4,5], which is defined for each region as  $|A_c - A_s|/A_s$ , where  $A_c$  is the area of the region in the cartogram and  $A_s$  is the specified area of that region, given by the geographic variable to be shown. The following list summarizes all quality criteria:

- Average and maximum cartographic error.
- Correct adjacencies of the rectangles (e.g., the rectangles for Germany and France should be adjacent, and the rectangles for Germany and Spain must not be adjacent).
- Maximum aspect ratio.
- Suitable relative positions (e.g., the rectangle for The Netherlands should be West of the one for Germany).

For a purely rectangular cartogram we cannot expect to simultaneously satisfy all criteria well. Figure 2 shows an example where correct adjacencies must be sacrificed in order to get a small cartographic error. In larger examples, bounding the aspect ratio of the rectangles also results in larger cartographic error.

30	10
10	30

**Fig. 2.** The values inside the rectangles indicate the preferred areas.

**Related work.** Rectangular cartograms are closely related to *floor plans* for electronic chips and architectural designs. Floor planning aims to represent a planar graph by its *rectangular dual*, defined as follows. A *rectangular partition* of a rectangle  $R$  is a partition of  $R$  into a set  $\mathcal{R}$  of non-overlapping rectangles such no four rectangles in  $\mathcal{R}$  meet at the same point. A *rectangular dual* of a planar graph  $(G, V)$  is a rectangular partition  $\mathcal{R}$ , such that (i) there is a one-to-one correspondence between the rectangles in  $\mathcal{R}$  and the nodes in  $G$ , and (ii) two rectangles in  $\mathcal{R}$  share a common boundary if and only if the corresponding nodes in  $G$  are connected. The following theorem was proven in [9]:

**Theorem 1.** *A planar graph  $G$  has a rectangular dual  $R$  with four rectangles on the boundary of  $R$  if and only if*

1. *every interior face is a triangle and the exterior face is a quadrangle*
2.  *$G$  has no separating triangles.*

Although every triangulated planar graph without separating triangles has a rectangular dual this does not imply that an error free cartogram for this graph exists. The area specification for every rectangle, as well as other criteria for good cartograms, may make it impossible to realize. Only a careful trade-off between the various types of errors (for example incorrect areas or incorrect adjacencies) makes it possible to visualize certain data sets as rectangular cartograms.

The only algorithm for standard cartograms that can be adapted to handle rectangular cartograms is Tobler's pseudo-cartogram algorithm [15] combined with a rectangular dual algorithm. Tobler's pseudo-cartogram algorithm starts with a orthogonal grid superimposed on the map, after which the horizontal and vertical lines of the grid are moved to stretch and shrink the strips in between. Hence, if the input to Tobler's algorithm is a rectangular partition, then the output is also a rectangular partition. However, Tobler's method is known to produce a large cartographic error and is mostly used as a preprocessing step for cartogram construction [10]. Furthermore, Tobler himself in a recent survey [14] states that none of the existing cartogram algorithms are capable of generating rectangular cartograms.

**Results.** We present the first fully automated algorithms for the computation of rectangular cartograms. We formalize the region adjacencies based on their geographic location and are so able to enumerate and process all feasible *rectangular layouts* for a particular subdivision (i.e., map). The precise steps that lead us from the input data to an algorithmically processable rectangular subdivision are sketched in Section 2.

We describe three algorithms that compute a cartogram from a rectangular layout. The first is an easy and efficient heuristic which we evaluated experimentally. The visually pleasing results of our implementation (which compare favorably with existing hand-drawn or computer assisted cartograms) can be found in Section 6. Secondly, we show how to formulate the computation of a cartogram as a bilinear programming problem (see Section 5).

For our third algorithm we introduce an effective generalization of sliceable layouts, namely *L-shape destructible* layouts. We prove in Section 3 that the coordinates of an L-shape destructible layout are uniquely determined by the specified areas (if an exact cartogram exists at all for a specific set of area values). The proof immediately implies an efficient algorithm to compute an exact (up to an arbitrarily small error) cartogram for subdivisions that arise from actual maps.

## 2 Algorithmic Outline

Assume that we are given an administrative subdivision into a set of regions. The regions and adjacencies can be represented by nodes and arcs of a graph  $F$ , which is the face graph of the subdivision.

**1. Preprocessing:** The face graph  $F$  is in most cases already triangulated (except for its outer face). In order to construct a rectangular dual of  $F$  we have to process internal nodes of degree less than four, and triangulate any remaining non-triangular faces. Our algorithms will consider every possible triangulation of non-triangular faces, since each may lead to a different rectangular dual.

**2. Directed edge labels:** Any two nodes in the face graph have at least one direction of adjacency which follows naturally from their geographic location (for example, The Netherlands lie clearly West of Germany). While in theory there

are four different directions of adjacency any two nodes can have, in practice only one or two directions are reasonable (for example, France can be considered to lie North or East of Spain).

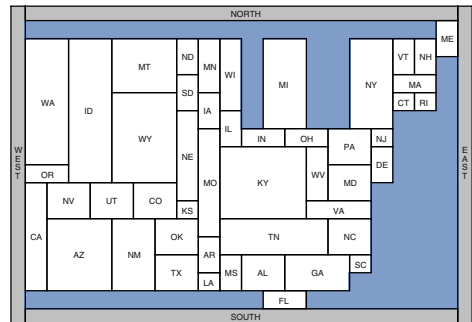
Our algorithms go through all possible combinations of direction assignments and determine which one gives a correct or the best result. While in theory there can be an exponential number of options, in practice there is often only one natural choice for the direction of adjacency between two regions. We call a particular choice of adjacency directions a *directed edge labeling*.

*Observation 1.* A face graph  $F$  with a directed edge labeling can be represented by a rectangular dual if and only if

1. every internal region has at least one North, one South, one East, and one West neighbor.
2. when traversing the neighbors of any node in clockwise order starting at the western most North neighbor we first encounter all North neighbors, then all East neighbors, then all South neighbors and finally all West neighbors.

A realizable directed edge labeling constitutes a *regular edge labeling* for  $F$  as defined in [6] which immediately implies our observation.

**3. Rectangular layout:** To actually represent a face graph together with a realizable directed edge labeling as a rectangular dual we have to pay special attention to the nodes on the outer face since they may miss neighbors in up to three directions. To compensate for that we add four special regions NORTH, EAST, SOUTH, and WEST, as well as *sea regions* that help to preserve the original outline of the subdivision. Then we can employ the algorithm by He and Kant [6] to construct a *rectangular layout*, i.e., the unique rectangular dual of a realizable directed edge labeling. The output of our implementation of the algorithm by He and Kant is shown in Figure 3.



**Fig. 3.** One of 4608 possible rectangular layouts of the US.

**4. Area assignment:** For a given set of area values and a given rectangular layout we would like to decide if an assignment of the area values to the regions is possible without destroying the correct adjacencies. Should the answer be negative or should the question be undecidable, then we still want to compute a cartogram that has a small cartographic error while maintaining reasonable aspect ratios and relative positions.

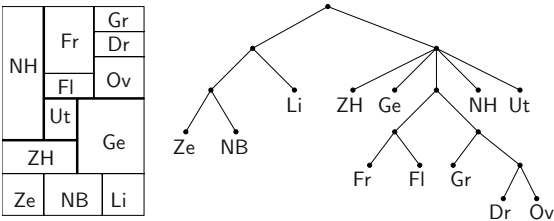
In this paper we introduce and discuss three methods for computing cartograms from rectangular layouts. We implemented (parts of) our algorithms and we present experimental results in Section 6. In the remainder of this section we give a short overview of our algorithms.



**Segment moving heuristic.** A simple but efficient heuristic: We iteratively move horizontal and vertical segments of the rectangular layout to reduce the maximum relative error. Additional details can be found in Sections 4 and 6.

**Bilinear programming.** The computation of a cartogram with the correct adjacencies and minimum relative error can be formulated as a *bilinear programming problem*, which unfortunately is nonconvex. Nevertheless, some non-linear programming methods may still be able to solve certain instances of the problem because the number of variables and constraints is only linear in the number of rectangles. Additional details can be found in Section 5.

**L-sequence algorithm.** For certain types of rectangular layouts we can compute an exact or nearly exact cartogram. For arbitrary layouts it is currently unknown if one can decide in polynomial time if an exact cartogram exists. We first determine for a given rectangular layout a *maximal rectangle hierarchy*. The maximal rectangle hierarchy groups rectangles that together form a larger rectangle, as illustrated in Figure 4. It can be computed in linear time [13]. All groups in the hierarchy are independent and we will determine areas separately for each group.

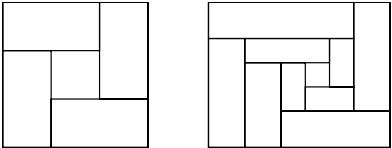


**Fig. 4.** Rectangular layout of the 12 provinces of the Netherlands and a corresponding maximal rectangle hierarchy.

A node of degree 2 in the hierarchy corresponds to a sliceable group of rectangles. If the maximal rectangle hierarchy consists of slicing cuts only, then we can (in a top-down manner) compute the unique position of each slicing cut. Here we may introduce wrong adjacencies, but this will only happen if there is no correct rectangular cartogram.

Nodes of a degree higher than 2 (which is necessarily at least 5) require more complex cuts (see for example the four thick segments in Fig. 4). One of the main contributions of this paper is the characterization of a type of non-sliceable layout for which the coordinates are still uniquely determined by the specified areas. We describe an efficient algorithm to compute a cartogram for *L-shape destructible* layouts (see Section 3 for a precise definition).

L-shape destructible layouts are a natural generalization of sliceable layouts with a clear practical value. For example, the rectangular layout of the 48



**Fig. 5.** Smallest non-sliceable layout (left), smallest non-L-shape destructible layout (right).

contiguous states of the US depicted in Figure 3 is not sliceable, but L-shape destructible. Figure 5 shows the smallest non-sliceable layout, as well as the smallest non-L-shape destructible layout.

In Section 3 we show that it is difficult to solve the computations for L-shape destructible layouts analytically. But it is possible to produce a cartogram based on one guessed value (coordinate) and decide whether the initial choice was too large or too small in linear time.

### 3 L-Shape Destructible Layouts

In this section we show that for certain non-sliceable rectangular layouts we can determine an exact or nearly exact rectangular cartogram efficiently (if one exists). Recall that a rectangular layout is a partition of a rectangle  $R$  into a set  $\mathcal{R}$  of non-overlapping rectangles. We call a rectangular layout  $\mathcal{R}$  *irreducible* if no proper subset of  $\mathcal{R}$  (of size  $> 1$ ) forms a rectangle. Furthermore, we call a rectilinear simple polygon with at most 6 vertices *L-shaped*. We say that an L-shaped polygon is *rooted* at a vertex  $p$  if one of its convex vertices is  $p$ .

**Definition 1 (L-shape destructible).** *An irreducible rectangular layout  $\mathcal{R}$  of a rectangle  $R$  is L-shape destructible if there is a sequence starting at a corner  $s$  of  $R$  in which the rectangles of  $\mathcal{R}$  can be removed from  $R$  such that the remainder forms an L-shaped polygon rooted at the corner  $t$  of  $R$  opposite to  $s$  after each removal.*

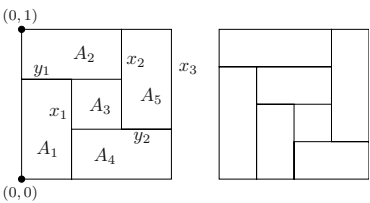
Note that the removal sequence is not necessarily unique. However, an incremental algorithm cannot make a bad decision, so the sequence can easily be computed in linear time. See Figure 6 for an example of a removal sequence for an irreducible layout obtained from the maximal rectangle hierarchy of the rectangular layout of the Eastern United States depicted in Figure 3.

The easiest type of non-sliceable but L-shape destructible layouts are *windmill* layouts (see Fig. 7 (left)). For a given set of area values  $\{A_1, \dots, A_5\}$  we can compute the (unique) realization analytically because this only requires finding roots of a polynomial of degree 2. We can scale the axes of any instance such that the outer rectangle has coordinates  $(0, 0)$ ,  $(0, 1)$ ,  $(A, 0)$ , and  $(A, 1)$ , where  $A = A_1 + A_2 + A_3 + A_4 + A_5$ . Then we get the equations  $x_1 \cdot y_1 = A_1$ ,  $x_2 \cdot (1 - y_1) = A_2$ ,  $(x_2 - x_1) \cdot (y_1 - y_2) = A_3$ ,  $(x_3 - x_1) \cdot y_2 = A_4$ , and  $(x_3 - x_2) \cdot (1 - y_2) = A_5$ , where also  $x_3 = A$ . Solving these equations with respect to, say,  $y_1$ , we obtain a quadratic equation in  $y_1$ . A windmill layout always has a realization as a cartogram, independent of the area values, which follows from the proof of Theorem 2.

Already for only slightly more complex types, for example Figure 7 (right), we cannot determine the coordinates exactly anymore, because this requires finding the roots of a polynomial of degree 6. Although the degree of the polynomials suggests that there may be more than one solution, we show that every L-shape destructible layout has at most one unique realization as a cartogram for a given set of area values. Furthermore we can compute the coordinates of this realization with an easy, iterative method.



**Fig. 6.** An L-shape destructible layout of the Eastern US. The shaded area shows the L-shaped polygon after the removal of rectangles 1 – 4.

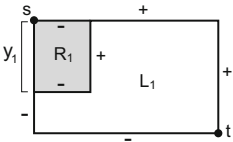


**Fig. 7.** A windmill layout (left) and a slightly more complex L-shape destructible layout (right).

**Theorem 2.** *An L-shape destructible layout with a given set of area values has exactly one or no realization as a cartogram.*

*Proof.* (sketch) W.l.o.g. we assume that the L-shape destruction sequence begins with a rectangle  $R_1$  that contains the upper left corner of  $R$ . We denote the height of  $R_1$  with  $y_1$  (see Fig. 8) and the L-shaped polygon that remains after the removal of  $R_1$  by  $L_1$ .

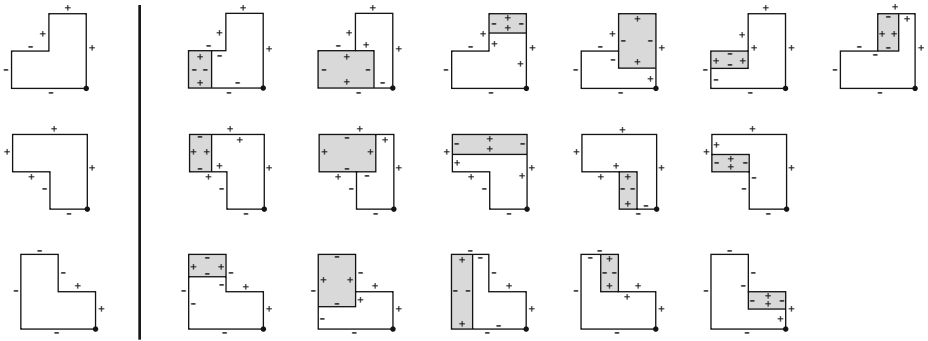
If we consider the edge lengths of  $L_1$  to be a function of  $y_1$  then we can observe that these functions are either monotonically increasing (denoted by a plus sign near the edge) or monotonically decreasing (denoted by a minus sign near the edge) in  $y_1$ . (Note here that the length of the right and bottom edges in Fig. 8 is actually fixed. We simply declare them to be monotonically increasing and decreasing respectively.) In fact, we will prove a much stronger statement: The lengths of the edges of the L-shaped polygons remaining after each removal of a rectangle in a L-shape reduction sequence are monotone functions in  $y_1$ .



**Fig. 8.** Setting the height of the first rectangle in an L-shape destruction sequence.

Clearly this statement is true after the removal of  $R_1$ .  $L_1$  is of the L-shape type depicted in the upper left corner of Figure 9. We will now argue by means of a complete case analysis that during a destruction sequence only L-shaped polygons of the three types depicted in the left column of Figure 9 can ever occur. Note that the type of a L-shaped polygon also describes which of its edges are monotonically increasing or decreasing in  $y_1$ .

For each type of L-shaped rectangle there is only a constant number of possible successors in the destruction sequence, all of which are depicted in Figure 9. We invite the reader to carefully study this figure to convince him or herself that it depicts indeed a complete coverage of all possibilities. Note that we never



**Fig. 9.** Removing a rectangle from a rooted L-shaped polygon. The plus and minus signs next to an edge indicate that its length is a monotonically increasing or monotonically decreasing function of  $y_1$ .

need to know exactly which functions describe the edge lengths of the L-shaped polygons - it is sufficient to know that they are always monotone in  $y_1$ .

During the destruction sequence it is possible that we realize that our initial choice of  $y_1$  was incorrect. For example, a rectangle  $R_i$  should be placed according to the layout as in the second case of the top row of Figure 9, but the area of  $R_i$  is too small and it would be placed as in the first case of the top row. Then the signs of the edges show that  $y_1$  must be increased to obtain correct adjacencies. Should no further increase of  $y_1$  be possible, then we know that the correct adjacencies can not be realized for this particular layout and area values.

As soon as the L-shaped polygon consists of only two rectangles we can make a decision concerning our initial choice of  $y_1$ . The edge lengths of one of the two rectangles are either both monotonically increasing or monotonically decreasing. A comparison of its current area with its prescribed area tells us if we have to increment or decrement  $y_1$ .

A binary search on  $y_1$  either leads to a unique solution or we recognize that the rectangular layout can not be realized as a cartogram with correct areas.  $\square$

## 4 Segment Moving Heuristic

A very simple but effective heuristic to obtain a rectangular cartogram from a rectangular layout is the following. Consider the maximal vertical segments and maximal horizontal segments in the layout, for example the vertical segment in Figure 3 that has Kentucky (KY) to its left and West Virginia (WV) and Virginia (VA) to its right. This segment can be moved a little to the left, making Kentucky smaller and the Virginias larger, or it can be moved to the right with the opposite effect.

The segment moving heuristic loops over all maximal segments and moves each with a small step in the direction that decreases the maximum error of the adjacent regions. After a number of iterations, one can expect that all maximal

segments have moved to a locally optimal position. However, we have no proof that the method goes to the global optimum, or that it even converges.

The segment moving heuristic has some important advantages: (i) it can be used for any rectangular layout, (ii) one iterative step for all maximal segments takes  $O(n)$  time, (iii) no area need to be specified for sea rectangles, (iv) a bound on the aspect ratio can be specified, and (v) adjacencies between the rectangles can be preserved, but need not be. Not preserving adjacencies can help to reduce cartographic error.

## 5 Bilinear Programming

Once a rectangular layout is fixed, the rectangular cartogram problem can be formulated as an optimization problem. The variables are the  $x$ -coordinates of the vertical segments and the  $y$ -coordinates of the horizontal segments. For proper rectangular layouts (no four-rectangle junctions, inside an outer rectangle) with  $n$  rectangles, there are  $n - 1$  variables. The area of each rectangle is determined by four of the variables.

We can formulate the minimum error cartogram problem as a *bilinear program*: the rectangle constraints for a rectangle  $R$  are:

$$\begin{aligned}(x_j - x_i) \cdot (y_l - y_k) &\geq (1 - \epsilon) \cdot A_R, \\ (x_j - x_i) \cdot (y_l - y_k) &\leq (1 + \epsilon) \cdot A_R\end{aligned}$$

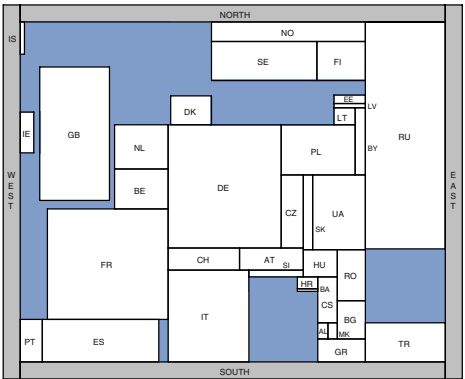
where  $x_i$  and  $x_j$  are the  $x$ -coordinates of the left and right segment,  $y_k$  and  $y_l$  are the  $y$ -coordinates of the bottom and top segment,  $A_R$  is the specified area of  $R$ , and  $\epsilon$  is the cartographic error for  $R$ . An additional  $O(n)$  linear constraints are needed to preserve the correct adjacencies of the rectangles (e.g., along sliceable cuts). Also, bounded aspect ratio (height-width) of all rectangles can be guaranteed with linear constraints. The objective function is to minimize  $\epsilon$ . The formulation is a quadratic programming problem with quadratic constraints that include non-convex ones, and a linear optimization function. Since there are no  $x_i^2$  or  $y_j^2$  terms, only  $x_i y_j$  terms, it is a bilinear programming problem. Several approaches exist that can handle such problems, but they do not have any guarantees [1].

## 6 Implementation and Experiments

We have implemented the segment moving heuristic and tested it on some data sets. The main objective was to discover whether rectangular cartograms with reasonably small cartographic error exist, given that they are rather restrictive in the possibilities to represent all rectangle areas correctly. Obviously, we can only answer this question if the segment moving heuristic actually finds a good cartogram if it exist. Secondary objectives of the experiments are to determine to what extent the cartographic error depends on maximum aspect ratio and

**Table 1.** Errors for different aspect ratios and sea percentages (correct adjacencies).

Data set	Sea	Asp.	Av. error	Max.
Eu elec.	20%	8	0.071	0.280
Eu elec.	20%	9	0.070	0.183
Eu elec.	20%	10	0.067	0.179
Eu elec.	20%	11	0.065	0.155
Eu elec.	20%	12	0.054	0.137
Eu elec.	10%	10	0.098	0.320
Eu elec.	15%	10	0.076	0.245
Eu elec.	20%	10	0.067	0.179
Eu elec.	25%	10	0.049	0.126



**Fig. 10.** A cartogram depicting the electricity production of Europe.

correct or false adjacencies. We were also interested in the dependency of the error on the outer shape, or, equivalently, the number of added sea rectangles.

Our layout data sets consist of the 12 provinces of the Netherlands (NL), 36 countries of Europe, and the 48 contiguous states of the USA. The results pertaining to the NL data set can be found in the full version of the paper. For Europe, we joined Belgium and Luxembourg, and Ukraine and Moldova, because rectangular duals do not exist if Luxembourg or Moldova are included as a separate country. Europe has 16 sea rectangles and the US data set has 9. For Europe we allowed 10 pairs of adjacent countries to be in different relative position, leading to 1024 possible layouts. Of these, 768 correspond to a realizable directed edge labeling. For the USA we have 13 pairs, 8192 possible layouts, and 4608 of these are realizable. In the experiments, all 768 or 4608 layouts are considered and the one giving the lowest average error is chosen as the cartogram.

As numeric data we considered for Europe the *population* and the *electricity production*, taken from [2]. For the USA we considered *population*, *native population*, number of *farms*, number of *electoral college votes*, and total length of *highways*. The data is provided by the US census bureau in the *Statistical Abstract of the United States*.<sup>1</sup>

Preliminary tests on all data sets showed that the false adjacency option always gives considerably lower error than correct adjacencies. The false adjacency option always allowed cartograms with average error of only a few percent. A small part of the errors is due to the discrete steps taken when moving the segments. Since cartograms are interpreted visually and show a global picture, errors of a few percent on the average are acceptable. Errors of a few percent are also present in standard, computer-generated contiguous cartograms [4,5,7, 8]. We note that most hand-made rectangular cartograms also have false adjacencies, and aspect ratios of more than 20 can be observed.

<sup>1</sup> <http://www.census.gov/statab/www/>

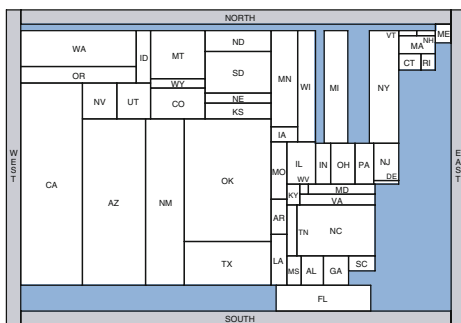
Table 1 shows errors for various settings for the electricity production data set. The rectangular layout chosen for the table is the one with lowest average error, the corresponding maximum error is shown only for completeness. In the table we observe that the error goes down with a larger aspect ratio, as expected. For Europe and population, errors below 10% on the average with correct adjacencies were only obtained for aspect ratios greater than 15. The table also shows that a larger sea percentage brings the error down. This is as expected because sea rectangles can grow or shrink to reduce the error of adjacent countries. More sea means more freedom to reduce error. However, sea rectangles may not become so small that they visually (nearly) disappear.

Table 2 shows errors for various settings for two US data sets. Again, we choose the rectangular layout giving the lowest average error. In the US highway data set, aspect ratios above 7 do not seem to decrease the error below a certain value. In the US population data set, correct adjacencies give a larger error than is acceptable. Even an aspect ratio of 40 gave an average error of over 0.3. We ran the same tests for the native population data and again observed that the error decreases with larger aspect ratio. An aspect ratio of 7 combined with false adjacency gives a cartogram with average error below 0.04 (see Fig. 11). Only the highways data allowed correct adjacencies, small aspect ratio, and small error simultaneously.

Figure 11 shows a rectangular cartogram of the US for one of the five data sets. Additional figures can be found in the full version of the paper. Three of them have false adjacencies, but we can observe that adjacencies are only slightly disturbed in all cases (which is the same as for hand-made rectangular cartograms). The data sets allowed aspect ratio of 10 or lower to yield an average error between 0.03 and 0.06, except for the farms data. Here an aspect ratio of 20 gives an average error of just below 0.1. Figures 1 and 10 show rectangular cartograms for Europe. The former has false adjacencies and aspect ratio bounded by 12, the latter has correct adjacencies and aspect ratio bounded by 8. The average error is roughly 0.06 in both cartograms.

**Table 2.** Errors for different aspect ratios, and correct/false adjacencies. Sea 20%.

Data set	Adj.	Asp.	Av. error	Max.
US pop.	false	8	0.104	0.278
US pop.	false	10	0.052	0.295
US pop.	false	12	0.022	0.056
US pop.	correct	12	0.327	0.618
US pop.	correct	14	0.317	0.612
US pop.	correct	16	0.308	0.612
US hw.	correct	6	0.073	0.188
US hw.	correct	8	0.058	0.101
US hw.	correct	10	0.058	0.101



**Fig. 11.** A cartogram depicting the native population of the United States.

## 7 Conclusion

In this paper we presented the first algorithms to compute rectangular cartograms. We showed how to formalize region adjacencies in order to generate algorithmically processable layouts. We also characterized the class of L-shape destructible layouts and showed how to efficiently generate exact or nearly exact cartograms for them. An interesting open question in this context is whether non-sliceable, not L-shape destructible layouts have a unique solution as well. Another open problem is whether rectangular cartogram construction (correct or minimum error) can be done in polynomial time.

We experimentally studied the quality of our segment moving heuristic and showed that it is very effective in producing aesthetic rectangular cartograms with only small cartographic error. Our tests show the dependency of the error on the aspect ratio, correct adjacencies, and sea percentage. Our implementation tests many different possible layouts, which helps considerably to achieve low error. We will extend our implementation and perform more tests in the near future to evaluate our methods further.

## References

1. M. Bazaraa, H. Sherali, and C. Shetty. *Nonlinear Programming - Theory and Algorithms*. John Wiley & Sons, Hoboken, NJ, 2nd edition, 1993.
2. *De Grote Bosatlas*. Wolters-Noordhoff, Groningen, 52nd edition, 2001.
3. B. Dent. *Cartography - thematic map design*. McGraw-Hill, 5th edition, 1999.
4. J. A. Dougenik, N. R. Chrisman, and D. R. Niemeyer. An algorithm to construct continuous area cartograms. *Professional Geographer*, 37:75–81, 1985.
5. H. Edelsbrunner and E. Waupotitsch. A combinatorial approach to cartograms. *Comput. Geom. Theory Appl.*, 7:343–360, 1997.
6. G. Kant and X. He. Regular edge labeling of 4-connected plane graphs and its applications in graph drawing problems. *Theor. Comp. Sci.*, 172:175–193, 1997.
7. D. Keim, S. North, and C. Panse. Cartodraw: A fast algorithm for generating contiguous cartograms. *IEEE Trans. Visu. and Comp. Graphics*, 10:95–110, 2004.
8. C. Kocmoud and D. House. A constraint-based approach to constructing continuous cartograms. In *Proc. Symp. Spatial Data Handling*, pages 236–246, 1998.
9. K. Koźmiński and E. Kinnen. Rectangular dual of planar graphs. *Networks*, 5:145–157, 1985.
10. NCGIA / USGS. Cartogram Central, 2002. [http://www.ncgia.ucsb.edu/projects/Cartogram\\_Central/index.html](http://www.ncgia.ucsb.edu/projects/Cartogram_Central/index.html).
11. J. Olson. Noncontiguous area cartograms. *Prof. Geographer*, 28:371–380, 1976.
12. E. Raisz. The rectangular statistical cartogram. *Geogr. Review*, 24:292–296, 1934.
13. S. Sur-Kolay and B. Bhattacharya. The cycle structure of channel graphs in non-slicable floorplans and a unified algorithm for feasible routing order. In *Proc. IEEE International Conference on Computer Design*, pages 524–527, 1991.
14. W. Tobler. Thirty-five years of computer cartograms. *Annals of the Assoc. American Cartographers*, 94(1):58–71, 2004.
15. W. Tobler. Pseudo-cartograms. *The American Cartographer*, 13:43–50, 1986.



# Multi-word Atomic Read/Write Registers on Multiprocessor Systems<sup>\*</sup>

Andreas Larsson, Anders Gidenstam, Phuong H. Ha, Marina Papatriantafilou,  
and Philippas Tsigas

Department of Comp. Science, Chalmers University of Technology, SE-412 96  
Göteborg, Sweden

**Abstract.** Modern multiprocessor systems offer advanced synchronization primitives, built in hardware, to support the development of efficient parallel algorithms. In this paper we develop a simple and efficient algorithm for atomic registers (variables) of arbitrary length. The simplicity and better complexity of the algorithm is achieved via the utilization of two such common synchronization primitives. In this paper we also evaluate the performance of our algorithm and the performance of a practical previously known algorithm that is based only on read and write primitives. The evaluation is performed on 3 well-known, parallel architectures. This evaluation clearly shows that both algorithms are practical and that as the size of the register increases our algorithm performs better, accordingly to its complexity behavior.

## 1 Introduction

In multiprocessing systems cooperating processes may share data via shared data objects. In this paper we are interested in designing and evaluating the performance of shared data objects for cooperative tasks in multiprocessor systems. More specifically we are interested in designing a practical wait-free algorithm for implementing registers (or memory words) of arbitrary length that could be read and written atomically. (Typical modern multiprocessor systems support words of 64-bit size.)

The most commonly required consistency guarantee for shared data objects is *atomicity*, also known as *linearizability*. An implementation of a shared object is *atomic* or *linearizable* if it guarantees that even when operations overlap in time, each of them appears to take effect at an atomic time instant which lies in its respective time duration, in a way that the effect of each operation is in agreement with the object's sequential specification. The latter means that if we speak of e.g. read/write objects, the value returned by each read equals the value written by the most recent write according to the sequence of “shrunk” operations in the time axis.

---

<sup>\*</sup> This work was supported by computational resources provided by the Swedish National Supercomputer Centre (NSC).

The classical, well-known and simplest solution for maintaining consistency of shared data objects enforces mutual exclusion. Mutual exclusion protects the consistency of the shared data by allowing only one process at time to access it. Mutual exclusion causes large performance degradation especially in multiprocessor systems [1] and suffers from potential priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [2].

Non-blocking implementation of shared data objects is an alternative approach for the problem of inter-task communication. Non-blocking mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. They offer significant advantages over lock-based schemes because i) they do not suffer from priority inversion; ii) they avoid lock convoys; iii) they provide high fault tolerance (processor failures will never corrupt shared data objects); and iv) they eliminate deadlock scenarios involving two or more tasks both waiting for locks held by the other.

Non-blocking algorithms can be lock-free or wait-free. *Lock-free* implementations guarantee that regardless of the contention and the interleaving of concurrent operations, at least one operation will always make progress. However, there is a risk that the progress of other operations might cause one specific operation to take unbounded time to finish. In a *wait-free* algorithm, every operation is guaranteed to finish in a limited number of steps, regardless of the actions of the concurrent operations. Non-blocking algorithms have been shown to be of big practical importance [3,4], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [5].

The problem of multi-word wait-free read/write registers is one of the well-studied problems in the area of non-blocking synchronization [6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]. The main goal of the algorithms in these results is to construct wait-free multiword read/write registers using single-word read/write registers and not other synchronization primitives which may be provided by the hardware in a system. This has been very significant, providing fundamental results in the area of wait-free synchronization, especially when we consider the nowadays well-known and well-studied hierarchy of shared data objects and their synchronization power [22]. A lot of these solutions also involve elegant and symmetric ideas and have formed the basis for further results in the area of non-blocking synchronization.

Our motivation for further studying this problem is as follows: As the aforementioned solutions were using only read/write registers as components, they necessarily have each write operation on the multi-word register write the new value in several copies (roughly speaking, as many copies as we have readers in the system), which may be costly. However, modern architectures provide hardware synchronization primitives stronger than atomic read/write registers, some of them even accessible at a constant cost-factor away from read/write accesses. We consider it a useful task to investigate how to use this power, to the benefit of designing economical solutions for the same problem, which can lead to struc-

tures that are more suitable in practice. To the best of our knowledge, none of the previous solutions have been implemented and evaluated on real systems.

In this paper we present a simple, efficient wait-free algorithm for implementing multi-word  $n$ -reader/single writer registers of arbitrary word length. In the new algorithm each multi-word write operation only needs to write the new value in one copy, thus having significantly less overhead. To achieve this, the algorithm uses synchronization primitives called *fetch-and-or* and *swap* [1], which are available in several modern processor architectures, to synchronize  $n$  readers and a writer accessing the register concurrently. Since the new algorithm is wait-free, it provides high parallelism for the accesses to the multi-word register and thus significantly improves performance. We compare the new algorithm with the wait-free one in [23], which is also practical and simple, and two lock-based algorithms, one using a single spin-lock and one using a readers-writer spin-lock. We design benchmarks to test them on three different architectures: UMA Sun-Fire-880 with 6 processors, ccNUMA SGI Origin 2000 with 29 processors and ccNUMA SGI Origin 3800 with 128 processors.

The rest of this paper is organized as follows. In Section 2 we describe the formal requirements of the problem and the related algorithms that we are using in the evaluation study. Section 3 presents our protocol. In Section 4, we give the proof of correctness of the new protocol. Section 5 is devoted to the performance evaluation. The paper concludes with Section 6, with a discussion on the contributed results and further research issues.

## 2 Background

**System and Problem Model.** A *shared register* of arbitrary length [23,24] is an abstract data structure that is shared by a number of concurrent processes which perform read or write operations on the shared register. In this paper we make no assumption about the relative speed of the processes, i.e. the processes are asynchronous. One of the processes, the *writer*, executes write operations and all other processes, the *readers*, execute read operations on the shared register. Operations performed by the same process are assumed to execute sequentially.

An implementation of a register consists of: (i) *protocols* for executing the operations (read and write); (ii) a data structure consisting of shared *subregisters* and (iii) a set of initial values for these. The protocols for the operations consist of a sequence of operations on the subregisters, called *suboperations*. These suboperations are reads, writes or other atomic primitives, such as *fetch-and-or* or *swap*, which are either available directly on modern multiprocessor systems or can be implemented from other available synchronization primitives [22]. Furthermore, matching the capabilities of modern multiprocessor systems, the subregisters are assumed to be atomic and to support multiple processes.

A register implementation is *wait-free* [22] if it guarantees that any process will complete each operation in a finite number of steps (suboperations) regardless of the execution speeds of the other processes.

For each operation  $O$  there exists a time interval  $[s_O, f_O]$  called its *duration*, where  $s_O$  and  $f_O$  are the starting and ending times, respectively. We assume that there is an precedence relation on the operations that form a strict partial order (denoted ' $\rightarrow$ '). For two operations  $a$  and  $b$ ,  $a \rightarrow b$  means that operation  $a$  ended before operation  $b$  started. If two operations are incomparable under  $\rightarrow$ , they are said to *overlap*.

A *reading function*  $\pi$  for a register is a function that assigns a high-level write operation  $w$  to each high-level read operation  $r$  such that the value returned by  $r$  is the value that was written by  $w$  (i.e.  $\pi(r)$  is the write operation that wrote the value that the read operation  $r$  read and returned).

**Criterion 1.** *A shared register is atomic iff the following three conditions hold for all possible executions:*

1. **No-irrelevant.** *There exists no read  $r$  such that  $r \rightarrow \pi(r)$ .*
2. **No-past.** *There exists no read  $r$  and write  $w$  such that  $\pi(r) \rightarrow w \rightarrow r$ .*
3. **No N-O inversion.** *There exist no reads  $r_1$  and  $r_2$  such that  $r_1 \rightarrow r_2$  and  $\pi(r_2) \rightarrow \pi(r_1)$ .*

**Peterson's Shared Multi-Word Register.** In [23] Peterson describes an implementation of an atomic shared multi-word register for one writer and many readers. The protocol does not use any other atomic suboperations than reads and writes and is described below.

The idea is to use  $n + 2$  shared buffers, which each can hold a value of the register, together with a set of shared handshake variables to make sure that the writer does not overwrite a buffer that is being read by some reader and that each reader chooses a stable but up-to-date buffer to read from. The shared variables are shown in Fig. 1 and the protocols for the read and write operations are shown in Fig. 2.

Peterson's implementation is simple and efficient in most cases, however, a high-level write operation potentially has to write  $n + 2$  copies of the new value and all high-level reads read at least two copies of the value, which can be quite expensive when the register is large. Our new register implementation uses these additional suboperations to implement high-level read and write operations that only need to read or write one copy of the register value.

We have decided to compare our method with this algorithm because: (i) they are both designed for the 1-writer  $n$ -reader shared register problem; (ii) compared to other more general solutions based on weaker subregisters (which are much weaker than what common multiprocessor machines provide) this one involves the least communication overhead among the processes, without requiring unbounded timestamps or methods to bound the unbounded version .

**Mutual-Exclusion Based Solutions.** For comparison we also evaluate the performance of two mutual-exclusion-based register implementations, one that uses a single *spin-lock* with exponential back-off and another that uses a *readers-writers spin-lock* [1] with exponential back-off to protect the shared register. The readers-writers spin-lock is similar to the spin-lock but allows readers to access the register concurrently with other readers.

Variable	Type	Description
WFLAG	Boolean	Indicates that the writer is writing in BUF1.
SWITCH	Boolean	To check if the writer has written in BUF1.
READING	Array of $n$ Boolean	Used together with WRITING to handle concurrent reads.
WRITING	Array of $n$ Boolean	Used together with READING to handle concurrent reads.
BUF1	Buffer	Main buffer.
BUF2	Buffer	Backup buffer.
COPYBUF	Array of $n$ buffers	An individual buffer copy for each reader.

**Fig. 1.** The shared variables used by Peterson’s algorithm. The number of readers is  $n$ . BUF1 holds the initial register value. All other variables are initialized to 0 or false.

<pre>Read operation by reader r. READING[r] = !WRITING[r]; flag1 = WFLAG; sw1 = SWITCH; read BUF1; flag2 = WFLAG; sw2 = SWITCH; read BUF1; bad1 = (sw1 != sw2)    flag1    flag2; if(READING[r] == WRITING[r]) {     return the value in COPYBUF[r]; } else if(bad1) {     return the value read from BUF2; } else {     return the value read from BUF1; }</pre>	<pre>Write operation. WFLAG = true; write to BUF1; SWITCH = !switch; WFLAG = false; for(each reader r) {     if(READING[r] != WRITING[r]) {         write to COPYBUF[r];         WRITING[r] = READING[r];     } } write to BUF2;</pre>
---	--

**Fig. 2.** Peterson’s algorithm. Lower-case variables are local variables.

### 3 The New Algorithm

The idea of the new algorithm is to remove the need for reading and writing several buffers during read and write operations by utilizing the atomic synchronization primitives available on modern multiprocessor systems. These primitives are used for the communication between the readers and the writer. The new algorithm uses  $n + 2$  shared buffers that each can hold a value of the register. The number of buffers is the same as for Peterson’s algorithm which matches the lower bound on the required number of buffers. The number of buffers cannot be less for any wait-free implementation since each of the  $n$  readers may be reading from one buffer concurrently with a write, and the write should not overwrite the last written value (since one of the readers might start to read again before the new value is completely written).

The shared variables used by the algorithm are presented in Fig. 3. The shared buffers are in the  $(n + 2)$ -element array BUF. The atomic variable SYNC is used to synchronize the readers and the writer. This variable consists of two fields: (i) the *pointer field*, which contains the index of the buffer in BUF that contains the most recent value written to the register and (ii) the *reading-bit field*,

Constants	Description
PTRFIELDLEN	The number of bits used for the pointer field indexing the most recently update buffer.
PTRFIELD	A bitmask containing 1's in the PTRFIELDLEN least significant bits.

Variable	Type	Description
SYNC	Unsigned word	Consists of two fields.
SYNC & PTRFIELD		Index of the buffer with the most recent register value.
bit PTRFIELDLEN + r of SYNC		The reading bit for reader r.
BUF	Array of n+2 buffers	The buffers for the register value.

**Fig. 3.** The constants and shared variables used by the new algorithm. The number of readers is  $n$ . Initially  $\text{BUF}[0]$  holds the register value and  $\text{SYNC}$  points to this buffer while all reader-bits are 0.

Read operation by reader $r$ .
--------------------------------

```

R1  readerbit = 1 << (r + PTRFIELDLEN)
R2  rsync = fetch_and_or(&SYNC,readerbit)
R3  rptr = rsync + PTRFIELD
R4  read(BUF[rptr])

```

Write operation.

```

W1  choose newwptr such that newwptr != oldwptr and
                                newwptr != trace[r] for all r;
W2  write(BUF[newwptr]);
W3  wsync = swap(&SYNC, 0 | newwptr); /* Clears all reading bits */
W4  oldwptr = wsync & PTRFIELD;
W5  for each reader r {
W6      if (wsync & (1 << (r + PTRFIELDLEN))) {
W7          trace[r] = oldwptr;
W8      }
W9  }

```

**Fig. 4.** The read and write operations of the new algorithm. The trace-array and oldwptr are static, i.e. stay intact between write operations. They are all initialized to zero.

which holds a *handshake* bit for each reader that is set when the corresponding reader has followed the value contained in the pointer field.

A reader (Fig. 4) uses *fetch-and-or* to atomically read the value of **SYNC** and set its reading-bit. Then it reads the value from the buffer pointed to by the pointer field.

The writer (Fig. 4) needs to keep track of the buffers that are available for use. To do this it stores the index of the buffer where it last saw each reader, in a  $n$ -element array `trace`, in persistent local memory. At the beginning of each write the writer selects a buffer index to write to. This buffer should be different from the last one it used and with no reader intending to use it. The writer writes the new value to that buffer and then uses the suboperation *swap* to atomically read `SYNC` and update it with the new buffer index and clear the reading-bits. The old value read from `SYNC` is then used to update the `trace` array for those readers whose reading-bit was set.

The maximum number of readers is limited by the size of the words that the two atomic primitives used can handle. If we are limited to 64-bit words we can support 58 readers as 6 bits are needed for the pointer field to be able to distinguish between  $58+2$  buffers.

## 4 Analysis

We first prove that the new algorithm satisfies the conditions in Lamport's criterion [12] (cf. Criterion 1 in section 2), which guarantee atomicity.

**Lemma 1.** *The new algorithm satisfies condition “No-irrelevant”*

*Proof.* A read  $r$  reads the value that is written by the write  $\pi(r)$ . Therefore  $r$ 's  $read(BUF[j])$  operation (line R4 in Fig. 4) starts after  $\pi(r)$ 's  $write(BUF[j])$  operation (line W2 in Fig. 4) starts. On the other hand, the starting time-point of the  $read(BUF[j])$  operation is before the ending time-point of  $r$  and the starting time-point of the  $write(BUF[j])$  operation is after the starting time-point of  $\pi(r)$ , so the ending time-point of  $r$  must be after the starting time-point of  $\pi(r)$ , or  $r \not\rightarrow \pi(r)$ .  $\square$

**Lemma 2.** *The new algorithm satisfies condition “No-past”*

*Proof.* We prove the lemma by contradiction. Assume there are a read  $r$  and a write  $w$  such that  $\pi(r) \rightarrow w \rightarrow r$ , which means i) write  $\pi(r)$  ends before write  $w$  starts and write  $w$  ends before read  $r$  starts and ii)  $r$  reads the value written by  $\pi(r)$ . Because  $\pi(r) \rightarrow w \rightarrow r$ , the value of  $SYNC$  that  $r$  reads (line R2 in Fig. 4) is written by a write  $w'$  using the *swap* primitive (line W3 in Fig. 4), where  $w' = w$  or  $w \rightarrow w' \rightarrow r$ , i.e.  $w' \neq \pi(r)$  because  $\pi(r) \rightarrow w$ . On the other hand, because  $r$  reads the buffer that is pointed by  $SYNC$  (lines R2-R4),  $r$  would read the buffer that has been written completely by  $w' \neq \pi(r)$ . That means  $r$  does not read the value written by  $\pi(r)$ , a contradiction.  $\square$

**Lemma 3.** *The new algorithm satisfies condition “No N-O inversion”*

*Proof.* We prove the lemma by contradiction. Assume there are reads  $r_1$  and  $r_2$  such that  $r_1 \rightarrow r_2$  and  $\pi(r_2) \rightarrow \pi(r_1)$ . Because i)  $\pi(r_1)$  always keeps track of which buffers are used by readers in the array *trace[]* (lines W4-W7 in Fig. 4) and ii) the buffer  $\pi(r_1)$  chooses to write to is different from those recorded in *trace[]* as well as the last buffer the writer has written, *wptr* (lines W1-W2),  $r_1$  reads the value written by  $\pi(r_1)$  only if  $r_1$  has read the correct value of  $SYNC$  (line R2 in Fig. 4) that has been written by  $\pi(r_1)$  (line W3 in Fig. 4).

On the other hand, because  $r_1 \rightarrow r_2$ , the value of  $SYNC$  that  $r_2$  reads must be written by a write  $w_k$  where  $w_k = \pi(r_1)$  or  $\pi(r_1) \rightarrow w_k$ , i.e.  $w_k \neq \pi(r_2)$  because  $\pi(r_2) \rightarrow \pi(r_1)$ . Moreover, because  $r_2$  reads the buffer that is pointed by  $SYNC$  (lines R2-R4),  $r_2$  would read the buffer that has been written completely by  $w_k \neq \pi(r_2)$  (lines W2-W3). That means  $r_2$  reads the value that has not been written by  $\pi(r_2)$ , a contradiction.  $\square$

**Complexity.** The complexity of a read operation is of order  $O(m)$ , where  $m$  is the size of the register, for both the new algorithm and Peterson's algorithm [23]. However, in Peterson's algorithm the reader may have to read the value up to 3 times, while in our algorithm the reader will only read the value once and has to use the *fetch-and-or* suboperation. A write operation in the new algorithm writes one value of size  $m$  and then traces the  $n$  readers. The complexity of the write operation is therefore of order  $O(n + m)$ . For Peterson's algorithm however, the writer must in the worst case write to  $n + 2$  buffers of size  $m$ , thus its complexity is of order  $O(n \cdot m)$ . As the size of registers and the number of threads increase, the new algorithm is expected to perform significantly better than Peterson's algorithm with respect to the writer. With respect to the readers the handshake mechanism used in the new algorithm can be more expensive compared to the one used by Peterson's, but on the other hand the new algorithm only needs to read one  $m$ -word buffer, whereas Peterson's need to read at least two and sometimes three buffers.

Using the above, we have the following theorem:

**Theorem 1.** *A multi-reader, single-writer,  $m$ -word sized register can be constructed using  $n + 2$  buffers of size  $m$  each. The complexity of a read operation is  $O(m)$ . The complexity of a write operation is  $O(n + m)$ .*

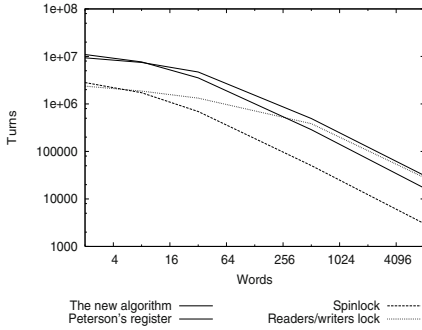
## 5 Performance Evaluation

The performance of the proposed algorithm was tested against: i) Peterson's algorithm [23], ii) a spinlock-based implementation with exponential backoff and iii) a readers-writers spinlock with an exponential backoff [1].

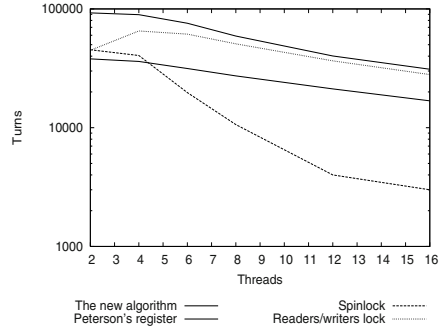
**Method.** We measured the number of successful read and write operations during a fixed period of time. The higher this number the better the performance. In each test one thread is the writer and the rest of the threads are the readers. Two sets of tests have been done: (i) one set with *low contention* and (ii) one set with *high contention*. During the high-contention tests each thread reads or writes continuously with no delay between successive accesses to the multi-word register. During the low-contention tests each thread waits for a time-interval between successive accesses to the multi-word register. This time interval is much longer than the time used by one write or read. Tests have been performed for different number of threads and for different sizes of the register.

**Systems.** The performance of the new algorithm has been measured on both UMA (Uniform Memory Architecture) and NUMA (Non Uniform Memory Architecture) multiprocessor systems. The difference between UMA and NUMA is how the memory is accessed. In a UMA system all processors have the same latency and bandwidth to the memory. In a NUMA system, processors are placed in nodes and each node has some of the memory directly attached to it. The processors of one node have fast access the memory attached to that node, but accesses to memory on another node has to be made over a network and is therefore significantly slower. The three different systems used are:

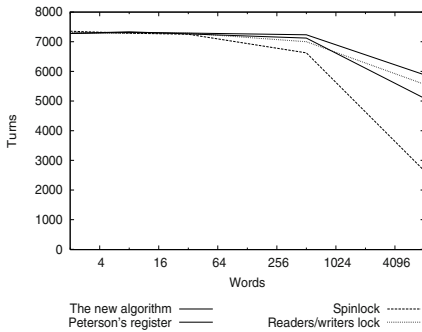




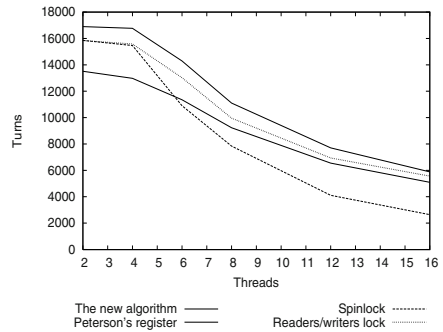
(a) 16 threads, high contention.



(b) 8192 words, high contention.



(c) 16 threads, low contention.

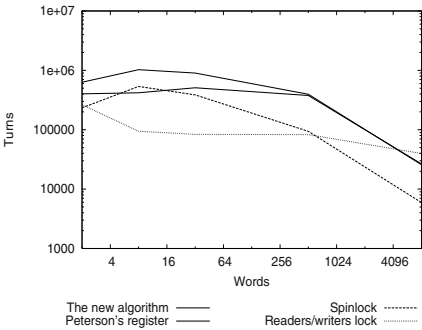


(d) 8192 words, low contention.

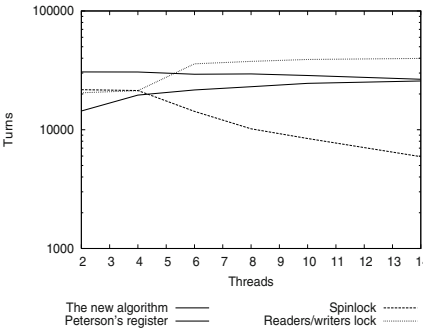
**Fig. 5.** Average number of reads or writes per process on the UMA SunFire 880.

- An UMA Sun SunFire 880 with 6 900MHz UltraSPARC III+ (8MB L2 cache) processors running Solaris 9.
- A ccNUMA SGI Origin 2000 with 29 250MHz MIPS R10000 (4MB L2 cache) processors running IRIX 6.5.
- A ccNUMA SGI Origin 3800 with 128 500MHz MIPS R14000 (8MB L2 cache) processors running IRIX 6.5.

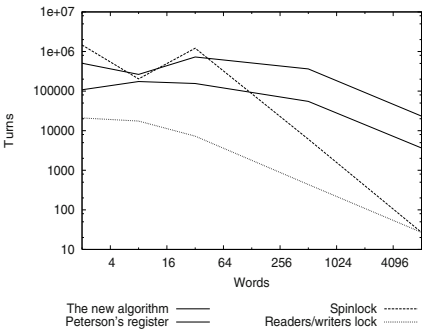
The systems were used non-exclusively, but for the SGI systems the batch-system guarantees that the required number of CPUs was available. The *swap* and *fetch-and-or* suboperations were implemented by the *swap* hardware instruction [25] and a lock-free subroutine using the *compare\_and\_swap* hardware instruction on the SunFire machine. On the SGI Origin machines *swap* and *fetch-and-or* were implemented by the *\_\_lock\_test\_and\_set* and the *\_\_fetch\_and\_or* synchronization primitives provided by the system [26], respectively.



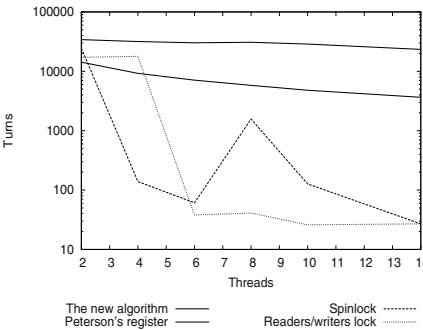
(a) All. 14 threads, high contention.



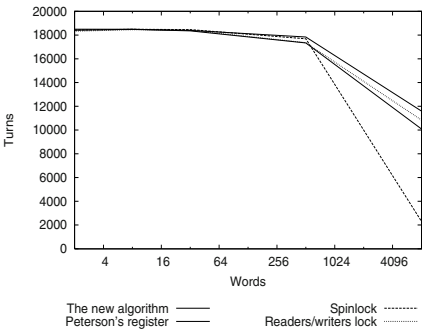
(b) All. 8192 words, high contention.



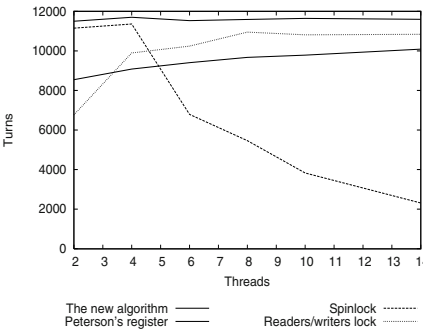
(c) Writer. 14 threads, high contention.



(d) Writer. 8192 words, high contention.

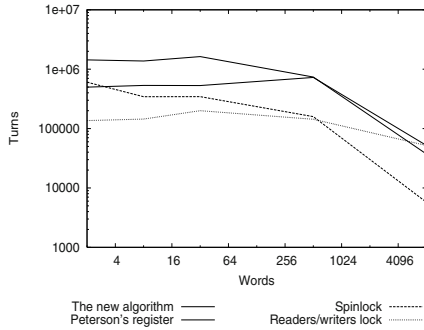


(e) All. 14 threads, low contention.

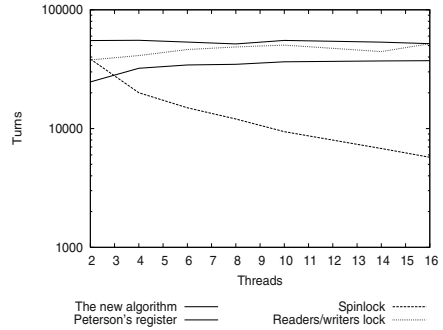


(f) All. 8192 words, low contention.

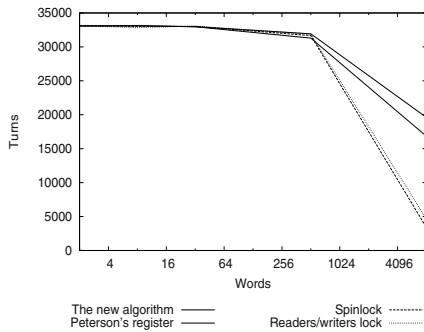
**Fig. 6.** Average number of reads or writes per process on NUMA Origin 2000



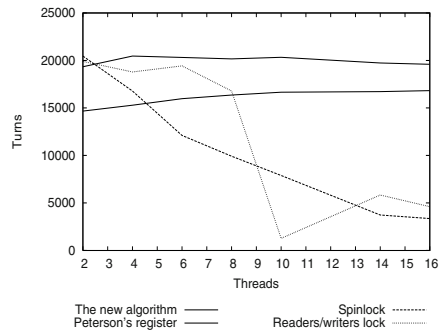
(a) 16 threads, high contention.



(b) 8192 words, high contention.



(c) 16 threads, low contention.



(d) 8192 words, low contention.

**Fig. 7.** Average number of reads or writes per process on NUMA Origin 3800.

**Results.** Following the analysis and the diagrams presenting the experiments' outcome, it is clear that the performance of the lock-based solutions is not even near the figures of the wait-free algorithms unless the number of threads is minimal (2) *and* the size of the register is small. Moreover, as expected following the analysis, the algorithm proposed in this paper performs at least as well and in the large-size register cases better than Peterson's wait-free solution.

More specifically, on the UMA SunFire the new algorithm outperforms the others for large registers under both low and high contention (cf. Fig. 5)

On the NUMA Origin 2000 and the NUMA Origin 3800 platforms (cf. Fig. 6 and 7, respectively), we observe the effect of the particular architecture, namely the possibility for creating contention on the synchronization variable may become high, affecting the performance of the solutions. By observing the performance diagrams for this case, we still see the writer in the new algorithm performs significantly better than the writer in Peterson's algorithm in both the

low and high-contention scenarios. Recall that the writer following Peterson's algorithm has to write to more buffers as the number of readers grow. The writer of the algorithm proposed here has no such problems. The latter phenomenon, though, has a positive side-effect in Peterson's algorithm: namely, as the writer becomes slower, the chances that the readers have to read their individual buffers apart from the regular two buffers, become smaller. Hence, the readers' performance difference of the two wait-free algorithms under high contention becomes smaller.

## 6 Conclusions

This paper presents a simple and efficient algorithm of atomic registers (memory words) of arbitrary size. The simplicity and the good time complexity of the algorithm are achieved via the use of two common synchronization primitives. The paper also presents a performance evaluation of i) the new algorithm; ii) a previously known practical algorithm that based only on read and write operations; and iii) two mutual-exclusion-based registers. The evaluation is performed on three different well-known multiprocessor systems.

Since shared objects are very commonly used in parallel/multithreaded applications, such results and further research along this line, on shared objects implementations, is significant towards providing better support for efficient synchronization and communication for these applications.

## References

1. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating Systems Concepts. Addison-Wesley (2001)
2. Sha, L., Rajkumar, R., Lojoczky, J.: Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers* **39** (1990) 1175–1185
3. Tsigas, P., Zhang, Y.: Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In: *Proc. of the ACM SIGMETRICS 2001/Performance 2001*, ACM press (2001) 320–321
4. Tsigas, P., Zhang, Y.: Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In: *Proc. of the 3rd ACM Workshop on Software and Performance (WOSP'02)*, ACM press (2002) 55–67
5. Sundell, H., Tsigas, P.: NOBLE: A non-blocking inter-process communication library. In: *Proc. of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*. LNCS, Springer Verlag (2002)
6. Bloom, B.: Constructing two-writer atomic registers. *IEEE Transactions on Computers* **37** (1988) 1506–1514
7. Burns, J.E., Peterson, G.L.: Constructing multi-reader atomic values from non-atomic values. In: *Proc. of the 6th Annual ACM Symposium on Principles of Distributed Computing*, ACM Press (1987) 222–231
8. Haldar, S., Vidyasankar, K.: Constructing 1-writer multireader multivalued atomic variable from regular variables. *Journal of the ACM* **42** (1995) 186–203
9. Haldar, S., Vidyasankar, K.: Simple extensions of 1-writer atomic variable constructions to multiwriter ones. *Acta Informatica* **33** (1996) 177–202

10. Israeli, A., Shaham, A.: Optimal multi-writer multi-reader atomic register. In: Proc. of the 11th Annual Symposium on Principles of Distributed Computing, ACM Press (1992) 71–82
11. Kirousis, L.M., Kranakis, E., Vitányi, P.M.B.: Atomic multireader register. In: Distributed Algorithms, 2nd International Workshop. Volume 312 of LNCS., Springer, 1988 (1987) 278–296
12. Lamport, L.: On interprocess communication. *Distributed Computing* **1** (1986) 77–101
13. Li, M., Vitányi, P.M.B.: Optimality of wait-free atomic multiwriter variables. *Information Processing Letters* **43** (1992) 107–112
14. Li, M., Tromp, J., Vitányi, P.M.B.: How to share concurrent wait-free variables. *Journal of the ACM* **43** (1996) 723–746
15. Peterson, G.L., Burns, J.E.: Concurrent reading while writing II: The multi-writer case. In: 28th Annual Symposium on Foundations of Computer Science, IEEE (1987) 383–392
16. Singh, A.K., Anderson, J.H., Gouda, M.G.: The elusive atomic register. *Journal of the ACM* **41** (1994) 311–339
17. Newman-Wolfe, R.: A protocol for wait-free, atomic, multi-reader shared variables. In: Proc. of the 6th Annual ACM Symposium on Principles of Distributed Computing, ACM Press (1987) 232–248
18. Vitányi, P.M.B., Awerbuch, B.: Atomic shared register access by asynchronous hardware. In: 27th Annual Symposium on Foundations of Computer Science, IEEE (1986) 233–243
19. Simpson, H.R.: Four-slot fully asynchronous communication mechanism. *IEE Proc., Computers and Digital Techniques* **137** (1990) 17–30
20. Chen, J., Burns, A.: A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical Report YCS-288, Department of Computer Science, University of York (1997)
21. Kopetz, H., Reisinge, J.: The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In: Proc. of the Real-Time Systems Symposium, IEEE Computer Society Press (1993) 131–137
22. Herlihy, M.: Wait-free synchronization. *ACM Transaction on Programming and Systems* **11** (1991) 124–149
23. Peterson, G.L.: Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems* **5** (1983) 46–55
24. Herlihy, M.: A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems* **15** (1993) 745–770
25. Weaver, D.L., Germond, T., eds.: *The SPARC Architecture Manual*. Prentice Hall (2000) Version 9.
26. Cortesi, D.: *Topics in IRIX Programming*. Silicon Graphics, Inc. (2004) (doc #:007-2478-009).

# Beyond Optimal Play in Two-Person-Zerosum Games

Ulf Lorenz

Department of Mathematics and Computer Science  
Paderborn University, Germany

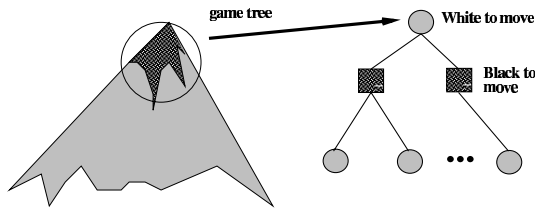
**Abstract.** Game tree search is the core of most attempts to make computers play games. Yet another approach is to store all possible positions in a database and to precompute the true values for all the positions. The databases allow computers to play optimally, in the sense that they will win every game once they reach a winning position. Moreover, they will never lose from a draw or win position. This is the situation that we find in games like Connect-4, in endgames of chess and in many other settings. Nevertheless, it has been observed that the programs do not play strongly when they have to play a tournament with strong, but non-perfect human players attending. In this paper we propose a mixture of game tree search and the database approach which attacks this problem on the basis of a known theoretical error analysis in game trees. Experiments show encouraging results.

## 1 Introduction

Game playing computer programs become stronger and stronger. One reason is that the search procedures of forecasting programs become faster and more sophisticated. Simultaneously, database approaches have been further developed. Some 6-men chess endgames — that are chess endgames with no more than 6 pieces on board — have been solved, and several easier games like Connect-4 as well. This means that for every position of the given game, the true value is known. Astonishingly, it is just this kind of perfection that leaves the mark of stupidity. When such a 'perfect' program reaches a winning position, it will always win, and when it reaches a draw position, it will never lose. Nevertheless, when it plays against fallible opponents, it is often not able to bring the opponent into trouble and to reach a better position than it had at the beginning. The programs cannot distinguish between 'easy' and 'difficult' positions for their opponents. This problem has rarely been tackled [14,7], and for the best of our knowledge never on the basis of a theoretical error analysis.

### 1.1 Game Tree Search Is an Error Filter

In this paper,  $G = (T, h)$  is a *game tree*, if  $T = (V, E)$  is a tree and  $h$  is a function, mapping nodes from  $V$  into  $\{-1, 0, 1\}$ . We identify the nodes of a game tree  $G$  with positions of the underlying game, and the edges of  $T$  with moves from one



**Fig. 1.** Only the (pre-)chosen partial tree is examined by a search algorithm.

position to the next. There are two players: MIN and MAX, and  $V$  can be split into two disjoint subsets of nodes. MAX makes moves at *MAX-nodes*, and MIN makes moves at *MIN-nodes*. In figures, MAX-nodes are symbolized by angular boxes. MIN-nodes are symbolized by circles. An algorithm which follows the *minimax principle* builds values of inner tree nodes with the help of *minimax values*. This means that when  $v$  is a MAX-node, the value of  $v$  is the maximum over the values of  $v$ 's successors. When  $v$  is a MIN-node, the value of  $v$  is the minimum over the values of  $v$ 's successors.

For most of the interesting board games, we do not know the correct values of all positions. Therefore, we are forced to base our decisions on heuristic or vague knowledge. Typically, a game playing program consists of three parts: a move generator, which computes all possible moves in a given position; an evaluation procedure which implements a human expert's knowledge about the value of a given position (these values are quite heuristic, fuzzy and limited) and a search algorithm, which organizes a forecast.

At some level of branching, the complete game tree (as defined by the rules of the game) is cut as shown in Figure 1, the artificial leaves of the resulting subtree are evaluated with the heuristic evaluations, and these values are propagated to the root of the game tree, as if they were real ones. For 2-person zero-sum games, computing this heuristic minimax value is by far the most successful approach in computer games history. When Shannon [18] proposed a design for a chess program in 1949 — which is in its core still used by all modern game playing programs — it seemed quite reasonable that deeper searches lead to better results. The astonishing observation over the last 40 years in the chess game and some other games is: the game tree acts as an error filter. The faster and more sophisticated the search algorithm, the better the search results!

Usually the tree to be searched is examined with the help of the Alphabeta algorithm [9] or the MTD(f)-algorithm [1]. In professional game playing programs, mostly the Negascout [15] variant of the Alphabeta algorithm is used. For theoretical examinations it is of minor importance which variant is used, because as far as the error frequency is concerned, it does not make any difference whether a pre-selected partial game tree is examined by the  $\alpha\beta$ -algorithm or by a pure minimax algorithm. In both cases, the result is the same. Only the effort to obtain the result differs drastically.

The approximation of the real root value with the help of fixed-depth searches leads to good results. Nevertheless, several enhancements that form the search tree more individually have been found. Some of these techniques are domain independent like Singular Extensions [4], Nullmoves [6], Conspiracy Number Searches [12] [16], and Controlled Conspiracy Number Search [10]. Many others are domain dependent. It is quite obvious that the form of the search tree strongly determines the quality of the search result.

## 1.2 Modeling the Mystery

In the eighties however, the first theoretical analyses suggested that searching deeper might lead to *worse* results. There have been papers by Pearl [13], G. Schröder [17], Althöfer [2,3], Kaibel and Scheucher [8] dealing with this error behavior. Lorenz and Monien [11] presented a one-to-one relation between the number of leaf-disjoint strategies and the robustness of game trees for minimax game trees with values of 0 and 1.

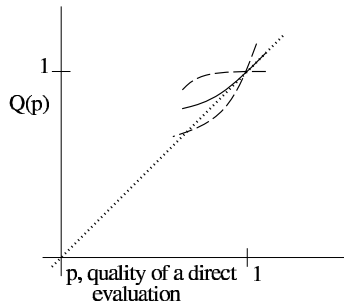
The newer papers have one idea in common: The authors assume that there is a reality, in that all positions of the game have an unequivocally true value. The art of a playing program, which can only deal with heuristic and vague value estimations, is to find out as much as possible of these true values. It is assumed that the heuristic evaluation procedure of the game playing programs is quite good in the sense that mostly the heuristic values are nearly correct, or that at least the relations between various values are more or less correct. The question is why and under which conditions the game tree helps or does not help the programs at their difficult task. One assumption might be that the ratio of inner tree nodes to game leaf nodes may vary, such that our definite knowledge about an end of a game (e.g. mate or stale mate) may help to improve the quality of the forecast. Another assumption might be that the quality of the programs' heuristic evaluation procedure increases with increasing depths. When we take a look at e.g. the chess game however, both assumptions seem unrealistic. When we are at move ten, it is not to see why the quality of our evaluation procedure should be better 20 plies away from the current position than only 16 plies away. And an examination of the search trees of the programs shows that it is neither correct that the number of mates and stale-mates increases with increasing depth. A third assumption is that it has to do with the structure of the game tree. The distribution of the possible true values over the nodes of the game tree might be a key. Indeed, this is exactly the assumption which we follow in this paper.

## 1.3 Modern Models and Results

The main result in [11] is the following. Three different measures for robustness are examined.

- A) An arbitrary but fixed finite game tree  $G$  is given. Each of its leaves has a 'true' value of 0 or 1. These values are propagated to inner nodes following the minimax principle, yielding true values for all nodes of the tree, in particular





**Fig. 2.** Possible courses of  $Q(p)$

the root. A second value is assigned to each leaf, called the heuristic value. With probability  $p$ , the heuristic and the true value of a leaf are equal. These values are propagated to the inner nodes according to the minimax principle as well. Denote by  $Q_G(p)$  the probability that the true and the heuristic value of the root of  $G$  are equal. If we denote by  $Q_G^{(n)}(\cdot)$  the  $n$ -th derivative of  $Q_G(\cdot)$ , the robustness is measured in the number of the first  $n$  derivations of  $Q_G(p)$  which are all zero at  $p$ . The motivation is as follows. Figure 2 presents us with three different courses of  $Q(p)$  and the identity function. The graphs of  $Q(p)$  can principally elapse in the three different ways, as shown in the figure. In the case of  $Q(p) \leq p$ , we do not see any motivation for performing any search at all. It is obviously more effective to directly evaluate the root. Thus, we see that the search tree is only useful (in an intuitive sense, here) when  $Q(p) > p$ , since only then is the error rate of the computed minimax value of the root smaller than the error rate of a direct evaluation of the root. Thus, if  $Q_G(p)$  and  $Q_H(p)$  are polynomials of quality and  $Q'_G(1) = 0$  and  $Q'_H(1) \geq 1$ , there will be an  $\epsilon > 0$  such that for all  $p \in [1 - \epsilon, 1]$  it is  $Q_G(p) > Q_H(p)$ , and thus  $G$  is more useful than  $H$ . The game trees, which are examined by iterative deepening search processes, form a sequence of supertrees resp. subtrees. It is now quite intuitive to demand that the error probability of supertrees should be above subtrees, at least starting with  $p \in [1 - \epsilon, 1]$ , in order to formalize what it means when we say that deeper search processes lead to better results. A point of criticism might be the fact that error- probabilities are assumed to be independent from each other and can be described by a constant  $p$ . This model however, can be seen as a special case of average case analysis.

Let us call an average-case error analysis, in which the number of occurring errors is weighted according to a binomial distribution, a *relaxed average-case analysis*. The term expresses that we are interested in the problem of how errors propagate in minimax trees when we presume an error rate of 'approximately  $x$  percent'.

Let  $G$  be an arbitrary game tree with  $n$  leaves, and let  $s$  be the string of 0s and 1s, consisting of the real values of the leaves of  $G$  in a left to right order. Furthermore, let  $s' \in \{0, 1\}^n$  be a further string of length  $n$ . If the

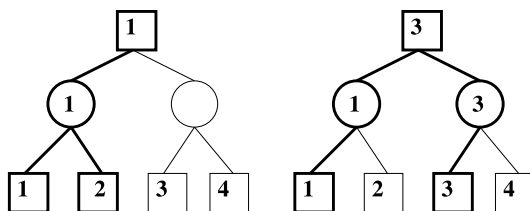


Fig. 3. A MAX- and a MIN-strategy

value  $s(i)$  and  $s(i')$  have the same value at a position  $i$ , we say 'the  $i$ -th leaf of  $s'$  has been correctly classified'. When we put all strings of length  $n$  into clusters  $C_1 \dots C_n$ , with each cluster containing those strings which have  $i$  correctly classified items, there exists a certain number  $c_i$  for each cluster of strings, which tells us in how many cases the root is correctly classified by a minimax-evaluation of  $G$ .

Since  $Q_G(p)$  is indeed equal to  $\sum_{i=0}^n \text{Prob}(\text{the root value is correctly classified by the minimax-evaluation of } G \mid \text{there are exactly } i \text{ correctly classified leaves}) \cdot \text{Prob}(\text{exactly } i \text{ leaves are correctly classified}) = \sum_{i=0}^n c_i / |C_i| \cdot \binom{n}{i} \cdot$

$x^i \cdot (1-x)^{n-i}$ , with  $x$  being equal to the probability  $p$  of our combinatorial model, the proposed model is nothing but a nice mathematical vehicle in which we perform a relaxed average-case error analysis. This means that it is difficult to place errors in a way that the given analysis does fail reality. Average case analyses are a common tool of analysis in computer science. Nevertheless, the question of whether or not a given average case analysis is well suited in order to analyze a given phenomenon, can only be proven by experiments.

- B) The robustness is the number of leaves whose value can be arbitrarily changed without the root value being effected. Thus, the robustness is the conspiracy number (minus 1) of the game tree and the root value, seen from the side of the true values.
- C) A *strategy* is a subtree of a given game tree with the following properties.
  - a) The root is part of the strategy.
  - b) For one of the players, exactly one successor of each node in the strategy is in the strategy as well. For the other player, all successors of the nodes in the strategy are always in it. A strategy looks as described in Figure 3. On the right, we see a so called MIN-strategy, which gives us, resp. proves an upper bound on the value of the root. On the left, the MAX-strategy proves a lower bound on the root value. Thus, a **strategy is quite a tiny sub-structure** of a game tree which gives us a lower or an upper bound on the root value. As we allowed only the values 0 and 1, a MIN-strategy with value 0 shows that the root value is 0, and a MAX-strategy with value 1 shows that the root value is 1. The robustness is measured in the number of leaf-disjoint strategies which give us either a lower bound 1 or an upper bound 0 on the root value.

**Theorem 1.** *The given three robustness notions are equivalent to each other.  $Q_G^{(k-1)}(1) = \dots = Q_G^{(1)}(1) = 0$  if and only if there are  $k$  leaf-disjoint strategies that prove the true value of the root, if and only if the values of at least  $k - 1$  arbitrarily chosen leaves can be arbitrarily changed without the true value of the root changing.*

In particular,  $k$  leaf-disjoint strategies ensure that  $Q_G(1-\varepsilon) \geq 1 - c\varepsilon^k$  for  $\varepsilon$  small,  $c$  a constant. Thus the number of leaf-disjoint strategies is a good indication for the robustness of a game tree. This was the first result for concrete (as opposed to randomly chosen, vague) game trees.

Doerr and Lorenz [5] extended the result to arbitrary leaf values and additionally showed that even for minimax -trees with arbitrary leaf values, it is valid that  $Q_G^{(k-1)}(1) = \dots = Q_G^{(1)}(1) = 0$  if and only one can arbitrarily change  $k - 1$  leaf values of  $G$  without the root value of  $G$  changing. Thus,  $k$  is the conspiracy number of  $G$ .

In this paper, we are going to present an algorithm which gets the perfect knowledge of a game database as input, and whose task it is to bring an opponent without perfect knowledge into trouble. The algorithm is based on the theoretical error analysis in game trees, as just presented. In the next section, we describe the algorithm, and in section 3 we present experimental data from its implementation.

## 2 Exploring the Local Game Tree Structure

Our aim is to find an algorithm which makes perfect moves only, and at the same time maximizes its opponent's chances to fail. As the aim is to make profit from the fact that some opponents may have no perfect knowledge about the game, we call it a *cheating algorithm*. Let  $v$  be a treenode with a value  $x$ . Let  $T$  be the smallest possible subtree of the allover game tree rooted at  $v$  which contains  $c$  leaf disjoint strategies which prove the value  $x$ . Let  $A$  be a search algorithm which tries to evaluate  $v$  with the help of search and with a heuristic evaluation function  $h : V \Rightarrow \mathbb{Z}$ . Let  $h$  be nearly perfect, making faults randomly distributed. Concerning the previous analysis, either  $A$  examines all nodes of  $T$ , or  $A$  wastes time because  $A$ 's probability to evaluate the node  $v$  correctly is not significantly higher than that of a static evaluation  $h(v)$ . Note that  $T$  can be arbitrarily large and arbitrarily irregular. Next, we are going to try to identify moves which lead into positions which are difficult to judge for every non-perfect opponent who tries to gain profit from a forecasting search.

Let us inspect an example presented in Figure 4. We assume that the true values of the game are  $-1, 0, 1$ . In level 0, at node  $v$ , we want to find a perfect move which is difficult to estimate for an arbitrary opponent. On level 1 our opponent has to move, and he will try to prove some move as the best one. Let us inspect the left branch of the root. The value of the node  $v_1$  is 0 and it has two successors with values 0, one with value 1. Our opponent is in danger to make a mistake when he either underestimates the two nodes  $v_4$  and  $v_5$ , or when

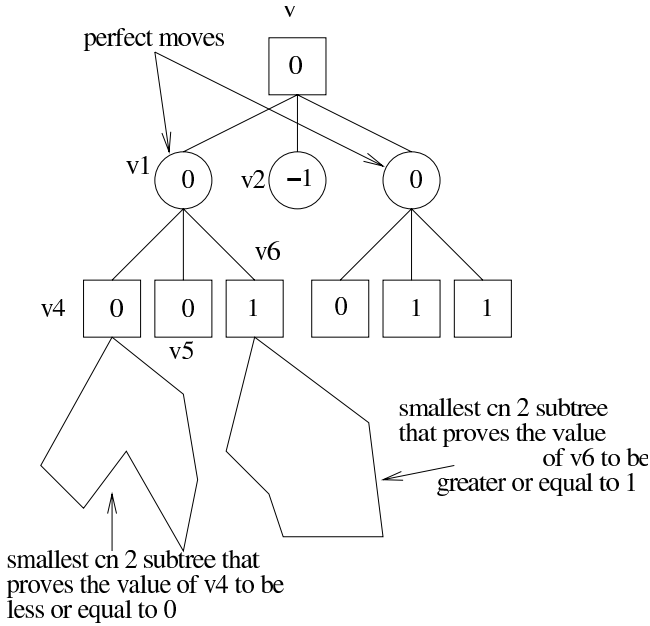


Fig. 4. The upper part of a game tree with a MAX-node at the root

he overestimates  $v_6$ . In order to make no mistake he must see that  $v_4$  or  $v_5$  is a good successor, and he must see that all successors with value  $-1$ , (i.e.  $v_6$ ) are bad for him. Let  $z_4$  be the number of nodes of the smallest tree below  $v_4$  that contains two leaf disjoint strategies, both showing that the value of  $v_4$  is less than or equal to 0. Let  $z_5$  be analogous for  $v_5$ . Let  $z_6$  be the number of nodes of the smallest tree below  $v_6$  that contains two leaf disjoint strategies, both showing that the value of  $v_6$  is greater than or equal to 1. We measure the difficulty of  $v_1$  as  $\min_{z_4, z_5} + z_6$ .

More generally: For simplification of the description, in the following let  $v$  be a MIN-successor of a MAX-root and  $v_1 \dots v_b$   $v$ 's successors. Let the value of the root be  $\leq 0$ , because in the case of root values  $> 0$  we play perfectly and win. In the following, a **cn-C tree rooted at node  $v$  and showing that  $f(v) > 0$  (resp.  $/ \geq, \leq, < 0$ )**, is a tree  $T$  which will prove the desired bound on the value of  $v$ , even if we arbitrarily change up to  $C - 1$  leaf values in  $T$ . We determine how difficult it is to correctly estimate the value of node  $v$  as follows.

function difficulty(MIN-node  $v$ , robustness  $C$ )

// let  $v$  be a MIN-node with value  $-1$  or  $0$

// let  $f$  be the function that maps nodes to its true values  $f(v)$ .

// let  $v_1, \dots, v_b$  be the successors of  $v$ .

if ( $f(v) == 0$ ) {

$cntSUM = \sum_{f(v_i) > 0}$  size of minimum cn-C tree rooted at  $v_i$  and  
showing that  $f(v_i) > 0$

```

    cntMIN = minimum $f(v_i) \leq 0$  size of minimum cn-C tree rooted at  $v_i$  and
               showing that  $f(v_i) \leq 0$ 
} else if ( $f(v) < 0$ ) {
    cntSUM =  $\sum_{f(v_i) \geq 0}$  size of minimum cn-C tree rooted at  $v_i$  and
               showing that  $f(v_i) \geq 0$ 
    cntMIN = minimum $f(v_i) < 0$  size of minimum cn-C tree rooted at  $v_i$  and
               showing that  $f(v_i) < 0$ 
}
return cntMIN + cntSUM;

```

Finding a cn-C tree is quite easy, and efficiently possible because it is based on strategies. We assign a so called target  $t_v = (c \in \mathbb{N}, b \in \{\leq, \geq, f(v)\})$  to the node  $v$ . If  $v$  is an ALL- node (i.e.  $v$  is MAX-node and  $b = \leq$ , or  $v$  is a MIN-node and  $b = \geq$ ) all successors of  $v$  must be examined and  $t_v$  is given to all these successors. Otherwise, if  $v$  is CUT-node,  $t_v$  can be split to those successors  $v_1 \dots v_d$  which are  $'b'f(v)$ . ('b' is a place holder for  $\leq$  or  $\geq$ .) Let  $t_i := (c_i, b_i, f(v))$ ,  $1 \leq i \leq d$  be the targets for all these successors  $v_1 \dots v_d$ . The split must be done such that  $\sum_{i=1}^d c_i$  equals  $c$ . If  $c = 1$  there is nothing else to do. We assume that every end of the game can correctly be recognized, such that when the end of the game is reached by mate, repetition or the 50-moves-rule, the target is fulfilled. Unfortunately, we need not any, but the smallest subtree with the desired properties, and there are a lot of possibilities for how to split a target. Nevertheless, all these possibilities can be examined with the help of a breadth-first-search, so that the effort is limited. In fact, this information could be pre-computed and incorporated into the database. For cn-2 the desired tree is even unique. For our experiments we used cn-2 and cn-3.

### 3 Experiment

We played an all-against-all chess tournament with 8 chess programs, the results of which are shown in Figure 5. The programs are 'R', 'CN2', 'CN3', 'HP', 'Shredder6', 'Fritz7', 'Gromit3.1', and 'SOS March 2000'. The last four programs are commercially available programs. 'Shredder6' and 'Fritz7' played with a fixed depth of 8, 'Gromit3.1' and 'SOS March 2000' only with a fixed depth of 4. These four programs have no database information. 'CN2' and 'CN3' are programs which follow the idea of the previous section. 'R' is a program which just randomly chooses one of the perfect moves. Thus, it is also a cn-1 program. 'HP' chooses also only perfect moves, but it does a depth-4 so called *prob-max* search. On even levels, it builds the value of a node  $v$  by taking the maximum over  $v'$ 's son values. On odd levels, it builds the value of a node  $v$  by taking the average over  $v'$ 's son values. The idea is taken from [7], but we did neither allow sub-perfect moves at the root, nor any domain specific knowledge.

We selected 20 endgame positions (Figure 6) as start-up positions and each program played 40 games (20 white and 20 black) against each other one. The 20 start-up positions are taken from endgame databases, such that the real values

The Tournament

	CN3	CN2	HP	Fritz	R	Shr.	Gromit	SOS	Σ	CN3Σ	CN2Σ	HPΣ	RΣ
CN3		20	20	23.5	20	25	25	25.5	159	99			
CN2	20		20	22.5	20	23.5	25	25	156		96		
HP	20	20		21.5	20	20.5	23	23	148			88	
Fritz	16.5	17.5	18.5		19.5	19.5	28	27	146.5	91	92	93	94
R	20	20	20	20.5		21.5	21.5	22	145.5				85.5
Shredder	15	16.5	19.5	20.5	18.5		25.5	28.5	144	89.5	91	94	93
Gromit	15	15.5	17	12	18.5	14.5		22	108.5	57.5	58	59.5	61
SOS	14.5	15	17	13	18	11.5	18		107	57	57.5	59.5	60.5

Fig. 5. Test Results

are known for all these positions, as well as for all possible sequences of moves starting from these positions up to the end of the game.

If you look at one line of Figure 5, you can see how a given program behaved against the others. For example, the CN3 program scored 23.5 point against Fritz, and it scored 159 points all together, as can be seen in the Σ-column. The four last columns show how the programs scored, under exclusion of three of the four database driven programs. For example, the CN3Σ-column shows how many points the programs scored in a sub-tournament, in which only the programs CN3, Fritz, Shredder, Gromit, and SOS participated.

The final ranking is strongly influenced by the selection of the start-up positions. If these positions are too difficult, none of the commercial programs can make profit from its positional knowledge. If they are too simple, no program fails. For example, the gaps between the programs might be made arbitrarily small, by just adding further simple drawing positions. Moreover, all positions are played with black and white by each program equally, so that it was not possible to score more than 50% against the database based programs. Nevertheless, we observe that 'CN2' never behaved worse than 'R' or 'HP', and 'CN3' never did worse than 'CN2', not even considering the games against one specific opponent. Thus, we see clear progress by the conspiracy numbers driven programs, and we see hardly any progress with the prob-max idea. When we examine the sub-tournaments with 'Shredder6', 'Fritz7', 'Gromit3.1', 'SOS March 2000' and only one database-based program at a time, we see that only 'CN2' and 'CN3' win their tournaments, while 'HP' and 'R' only reach rank 3. For example, you get 'CN3Σ' when you delete all games of CN2, HP and R from the above table and sum up the rest.

We might go even further and ask which program performs best in cheating 'SOS March 2000' (the least successful program concerning this little tournament). Let us play only half of the games, i.e. 'SOS' plays always white, with the exceptions of the positions 4,14, and 17 where it plays black. Then the ranking is Shredder (14 pts), Fritz (11.5 pts), CN3 and CN2 (10.5 pts each), Gromit (9 pts), HP (8.5 pts) and R (8 pts). You can, however, turn the numbers as you ever want, CN3 and CN2 are always much more successful than HP and R.

---

8/R7/2r5/5p2/6k1/8/5K2/8	b - - 0 1	KRkrp
8/6R1/8/5p2/6k1/2r5/5K2/8	b - - 0 8	KRkrp
8/6r1/8/4k3/8/5B2/5K2/8	w - - 0 8	KBkr
8/6r1/8/4k3/8/4BR2/5K2/8	b - - 0 8	KRBkr
7k/6p1/6q1/8/8/8/4QK2/8	w - - 0 8	KQkqp
6K1/3p4/3k4/8/N7/8/8/1N6	b - - 0 1	KNNkp
8/8/R7/8/k7/3P4/6b1/K7	b - - 0 1	KRPkb
8/7Q/7K/2r5/2p5/8/1k6/8	w - - 0 1	KQkrp
8/p7/8/4K3/7k/5b2/8/1R6	w - - 0 1	KRkbp
k1K5/2n5/8/8/b7/1R6/8/8	w - - 0 1	KRkbn
k7/p7/8/K7/R7/8/8/5n2	w - - 0 1	KRknp
K7/5b2/2R5/8/4k3/5R2/8/8	b - - 0 1	KRRkb
8/8/8/k1p5/2P5/1K6/P7/8	w - - 0 1	KPPkp
8/8/4p3/4k3/8/3PKP2/8/8	w - - 0 1	KPPkp
8/3K1kn1/2Q2b2/8/8/8/8/8	w - - 0 1	KQkbn
K7/P4k2/8/8/8/8/4R3/1r6	w - - 0 1	KRPkr
5k2/8/3KPP2/8/7n/8/8/8	b - - 0 1	KPPkn
K7/8/8/5pp1/2k2N2/8/8/8	w - - 0 1	KNkpp
8/KP1n4/2k5/8/5N2/8/8/8	w - - 0 1	KNPkn
8/1K6/P7/kn6/4N3/8/8/8	w - - 0 1	KNPkn

---

**Fig. 6.** The start positions in standard fen format

Last but not least, we present some observations made during the games:

a) The cn-based programs avoid move repetition draws, as those draws lead to small difficulties. b) They tend to keep the material on board, and they c) cover the idea of singular extensions. We found no simple way to characterize the differences between the CN2 and CN3 programs in the chess-language.

## 4 Summary and Conclusion

We presented an algorithm which has the perfect knowledge of a game database and which has been designed to bring the opponent into trouble. The algorithm is based on a theoretical error analysis in game trees. It has been implemented in a program which played a tournament against chess programs on a collection of start positions from endgame databases. No chess-dependent heuristic evaluation knowledge has been used. The program has shown that it makes clear progress with the help of inspecting the tree structure.

## References

1. A. de Bruin A. Plaat, J. Schaeffer and W. Pijls. A minimax Algorithm better than SSS\*. *Artificial Intelligence*, 87:255–293, 1999.
2. I. Althöfer. Root evaluation errors: How they arise and propagate. *ICCA Journal*, 11(3):55–63, 1988.

3. I. Althöfer. Generalized minmax algorithms are no better error correctors than minmax itself. In D.F. Beal, editor, *Advances in Computer Chess 5*, pages 265–282. North-Holland, 1990.
4. T.S. Anantharaman. Extension heuristics. *ICCA Journal*, 14(2):47–63, 1991.
5. B. Doerr and U. Lorenz. Error propagation in game trees. *submitted to CG'04*, 2004.
6. C. Donninger. Null move and deep search. *ICCA Journal*, 16(3):137–143, 1993.
7. P. J. Jansen. Kqkr: Speculativ thwarting a human opponent. *ICCA Journal*, 16(1):3–17, 1993.
8. H. Kaindl and A. Scheucher. The reason for the benefits of minmax search. In *Proc. of the 11<sup>th</sup> IJCAI*, pages 322–327, Detroit, MI, 1989.
9. D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
10. U. Lorenz. Controlled Conspiracy-2 Search. *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, (H. Reichel, S. Tison eds), Springer LNCS, pages 466–478, 2000.
11. U. Lorenz and B. Monien. The secret of selective game tree search, when using random-error evaluations. *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS)* (H. Alt, A. Ferreira eds), Springer LNCS, pages 203–214, 2002.
12. D.A. McAllester. Conspiracy Numbers for Min-Max searching. *Artificial Intelligence*, 35(1):287–310, 1988.
13. J. Pearl. *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Co., Reading, MA, 1984.
14. A. L. Reibman and B. W. Ballard. Non-Minimax Search Strategies for Use Against Fallible Opponents. *Proceedings of the National Conference on Artificial Intelligence, AAAI*, pages 338–342, 1983.
15. A. Reinefeld. *Spielbaum - Suchverfahren*. Springer, 1989.
16. J. Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, 1990.
17. G. Schröder. *Minimax-Suchen Kosten, Qualität und Algorithmen*. Doctoral-Thesis, University of Braunschweig, Germany, 1988.
18. C.E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, 1950.



# Solving Geometric Covering Problems by Data Reduction

Steffen Mecke and Dorothea Wagner\*

Dep. of Computer Science, University of Karlsruhe, {mecke, wagner}@ira.uka.de

**Abstract.** We consider a scenario where stops are to be placed along an already existing public transportation network in order to improve its attractiveness for the customers. The core problem is a geometric set covering problem which is  $\mathcal{NP}$ -hard in general. However, if the corresponding covering matrix has the consecutive ones property, it is solvable in polynomial time. In this paper, we present data reduction techniques for set covering and report on an experimental study considering real world data from railway systems as well as generated instances. The results show that data reduction works very well on instances that are in some sense “close” to having the consecutive ones property. In fact, the real world instances considered could be reduced significantly, in most cases even to triviality. The study also confirms and explains findings on similar techniques for related problems.

## 1 Introduction

The continuous stop location problem consists of placing new stations or stops along a given public transportation network in order to reach all potential customers and thus to improve attractiveness of using public means of transport. In particular, this scenario occurs in local public transport where additional stops in a rapid transit system or bus system, e.g. close to a residential or a settlement may induce significant increase of customers. Of course, the placement of new stops has certain additional effects to be considered as well. New stations require investment and operational costs and the travel time for customers already using a bus or train on the affected line might increase. On the other hand, the total number of customers and the according gain is improved and the total door-to-door travel time might be decreased.

As part of a project with the largest German railway company, we focused on a scenario where only the reachability of the network for potential customers was considered. That is, we are given a geometric graph, a (possibly empty) set of stations placed on the graph, a set of demand points in the plane and a radius. Weights are assigned to all edges and vertices of the graph representing the increase of travel time caused by new stop placed there. Then a demand point is covered by a (possibly new) station if the station is within the given radius

---

\* The authors acknowledge financial support from the EU for the RTN AMORE under grant HPRN-CT-1999-00104.

of that point. The problem consists in finding a set of stations with minimum total weight placed on the graph such that all demand points are covered. In [1] and [2] this problem is introduced as the continuous stop location problem and proved to be  $\mathcal{NP}$ -hard. It is also shown, that the problem can be formulated as a set covering problem by reducing the set of potential stations to a finite set of candidates.

Set covering problems have been studied widely, for example in the context of crew scheduling. See [3] for an overview. As shown for example in [4], the set covering problem is notoriously hard to solve or even to approximate. However, it is observed already in [1] that set covering is solvable in polynomial time by linear programming if the covering matrix has the so-called consecutive ones property. In the geometric setting, the special case that the network consists of only one single line satisfies the consecutive ones property. Polynomial solvability of some similar special cases of related problems is also shown in [5].

Experiments with railway data indicate that the size of such instances can be reduced significantly by applying a simple reduction technique mentioned already in [6]). Therefore we examine the power of data reduction for geometric set covering instances from the station location scenario as well as on generated instances. In addition to the known reduction techniques we apply a more advanced reduction rule and a refined enumeration algorithm to solve weighted set covering problems. It turns out that all real world instances could be reduced almost completely. Even the largest instances were reduced within one minute leaving an instance that could usually be solved within a few seconds.

One explanation for this nice behaviour is that the matrices occurring in our setting are close to having consecutive ones property or band diagonal form. In comparison, we conducted experiments on generated matrices (nearly) having consecutive ones property and found a similar behaviour. The computational results suggest that the effectiveness of data reduction highly depends on “closeness” of the matrix to having consecutive ones property. For the real world data considered, closeness to the consecutive ones property is obviously affected by the underlying geometry. The study also confirms and explains results reported in [7] about the power of data reduction for similar covering problems.

In the following section, well known reduction techniques for set covering are introduced and our new advanced reduction technique is presented. Efficient algorithms to implement these techniques are given and their behaviour is analyzed especially for instances with consecutive ones property. In Section 3 a refined enumeration algorithm for set covering is introduced and its running time is analyzed. Section 4 presents our experimental study and discusses the results.

## 2 Data Reduction for Set Covering

### 2.1 Basic Definitions

An *instance*  $I := (P, F, A, w)$  of our problem is given by two finite sets  $P$  and  $F$  with  $m := |P|$ ,  $n := |F|$ , a positive weight function  $w$  on  $F$  and a  $m \times n$ -matrix  $A = (a_{ij})_{i \in P, j \in F}$  over  $\{0, 1\}$ . We want to find a *covering* of  $P$  with

minimum weight, i.e. a subset  $F' \subseteq F$  such that for each  $p \in P$  there is  $f \in F'$  such that  $a_{pf} = 1$  and  $\text{cost}(F') := \sum_{f \in F'} w_f$  is minimized.

If  $a_{pf} = 1$  for a row  $p$  and a column  $f$  of  $A$ , then  $p$  is *covered* by  $f$ . For subsets  $F' \subseteq F$  and  $P' \subseteq P$  denote

$$\begin{aligned}\text{cov}(F') &:= \{p \in P \mid f \in F', a_{pf} = 1\} \\ \text{precov}(P') &:= \{f \in F \mid p \in P', a_{pf} = 1\} .\end{aligned}$$

A set  $F' \subseteq F$  such that  $\text{cov}(F') = P$  is called a *feasible solution* for  $I$ . It is called an *optimal solution* for  $I$  if  $\text{cost}(F')$  is minimum. The set of all optimal solutions will be denoted by  $\text{OPT}(I)$  and the weight of an optimal solution by  $\text{cost}(I)$ . An instance has a feasible solution if and only if every row of  $A$  contains at least one 1. We will assume in the following that all instances are solvable. As an instance of SET COVERING is fully described by its covering matrix  $A$  and the weight function  $w$ , we also write  $I = (A, w)$  as an abbreviation. In the following let  $N := |\{a_{ij} = 1\}|$ .

In the context of stop location  $P$  corresponds to the demand points in the plane (population areas) and  $F$  to the set of facilities (stations). Usually, an instance of the stop location problem is given by the geometric graph, the set of stations, the set of demand points in the plane and a radius. It is obvious that the covering matrix  $A$  can be easily derived from this. Moreover, we consider cases where no set of stations is given explicitly. Instead, a station can be placed on any (or given) positions on the graph. As already mentioned in the introduction, such an instance can be transformed into an equivalent instance with a finite set of stations.

**Definition 2.1.** A matrix over  $\{0, 1\}$  has the strong consecutive ones property, if the ones in every row are consecutive. It has the (simple) consecutive ones property (C1P) if its columns can be permuted such that the resulting matrix has the strong consecutive ones property.

For a matrix with C1P, a permutation of its columns to induce a matrix with strong consecutive ones property can be found in linear time using PQ-trees (cf. [8]).

## 2.2 Reduction

**Definition 2.2.** For columns  $f$  and  $g$  of  $A$ ,  $f$  is dominated by  $g$  if either  $\text{cov}(f) \subseteq \text{cov}(g)$  and  $w_f \geq w_g$  or  $\text{cov}(f) = \emptyset$ . For rows  $p$  and  $q$  of  $A$ ,  $p$  is dominated by  $q$  if  $\text{precov}(q) \subseteq \text{precov}(p)$ .

For an instance  $(P, F, w, A)$  we use the following terminology:

- If there are rows  $p \neq q$  such that row  $p$  is dominated by  $q$ , we call the instance *reducible by rows*. If there are columns  $f \neq g$  such that column  $f$  is dominated by  $g$ , we call the instance *reducible by columns*. An instance that is neither reducible by rows nor by columns is called *completely reduced*.

- A *reduction step* is denoted by a triple  $(M, i, k)$ , where  $M \in \{\text{row}, \text{col}\}$ ,  $i, k$  denote two rows (resp. columns) and  $i$  is dominated by  $k$ .
- Let  $\sigma = (\text{row}, i, k)$  (resp.  $\tau = (\text{col}, i, k)$ ) denote a reduction step. By  $\sigma A$  (resp.  $\tau A$ ) we denote the matrix resulting from deleting row (resp. column)  $i$ . The according instance is denoted by  $\sigma I$  (resp.  $\tau I$ ).

It is a well known fact (see e.g. [6]) that dominated rows and columns can be deleted without changing solvability. Even more, the value of an optimal solution is maintained. More precisely:

**Lemma 2.3.** *Let  $\sigma$  be a reduction step for an instance  $I$ . Then we have*

$$\text{cost}(\sigma I) = \text{cost}(I) \text{ and } \mathcal{OPT}(\sigma I) \subseteq \mathcal{OPT}(I) \text{ .}$$

Accordingly, in order to solve an instance of SET COVERING one can first apply a sequence of reduction steps and then solve the (completely) reduced instance. It can be even shown that a completely reduced instance is in a certain sense unique:

**Theorem 2.4.** *Let  $\mathcal{I} = (\mathcal{A}, \gamma)$  an instance and  $I = (A, c)$  and  $J = (B, d)$  two completely reduced instances derived from  $\mathcal{I}$ . Then we have, for a suitable permutation  $\pi$  of the rows and columns of  $A$*

$$\pi A = B \text{ and } \pi c = d \tag{1}$$

and thus

$$\text{cost}(I) = \text{cost}(\mathcal{I}) = \text{cost}(J) \tag{2}$$

$$\mathcal{OPT}(I) \subseteq \mathcal{OPT}(\mathcal{I}) \supseteq \mathcal{OPT}(J) \tag{3}$$

$$|\mathcal{OPT}(I)| = |\mathcal{OPT}(J)| \text{ .} \tag{4}$$

We omit a formal proof here. Note that (2) already follows from Lemma 2.3. Equation (1) follows from the fact that, for two different reduction steps  $\sigma$  and  $\tau$  for a matrix  $A$ , either  $\tau A = \sigma A$  or  $\tau$  is still a valid reduction step for  $\sigma A$ . Using this observation one can derive a bijection between the sequences of reduction steps leading to  $I$  and  $J$  respectively.  $\square$

### 2.3 Advanced Reduction

In many cases, a column is dominated in the sense that a combination of two or more other columns would dominate it. Consider a column  $f$  and a set of columns  $G$  such that  $\text{cov}(f) \subseteq \text{cov}(G)$  and

$$w_f \geq \sum_{g \in G} w_g \text{ .}$$

Then  $f$  is *dominated by  $G$* . If a column  $f$  is dominated by a set of columns  $G$ , we call its deletion *advanced reduction step*. However, it seems computationally

infeasible to consider for a column  $f$  all combinations of other columns. Instead we apply the following approach. For every row  $r$  and column  $f$  denote  $f_{\min}(r)$  a column in  $\text{precov}(r)$  with minimal weight. Then column  $f$  is dominated by  $g$  together with the set of columns  $\{f_{\min}(r) | r \in \text{cov}(f) - \text{cov}(g)\}$  if  $\text{cov}(f) \cap \text{cov}(g) \neq \emptyset$  and

$$w_f \geq w_g + \sum_{r \in \text{cov}(f) - \text{cov}(g)} w_{f_{\min}(r)} .$$

For this advanced reduction, analogous results to Lemma 2.3 and Theorem 2.4 can be proved. Note however, that by applying advanced reduction certain types of solutions might be lost. Consider for example the covering matrix  $\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$  with weights  $(2, 1, 1)$ . It is not reducible by simple reduction, optimal solutions are  $S_1 = \{1\}$  and  $S_2 = \{2, 3\}$ . By advanced reduction it can be reduced to  $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ , with only a two element optimal. In the next section we will show that the advanced reduction can be implemented without increasing the asymptotic worst-case running time of the reduction.

## 2.4 Reduction Algorithm

We shortly review the algorithmic realization of reduction. In the following we consider a matrix obtained by reductions consisting of reduction steps as well as advanced reduction steps.

**Theorem 2.5.** *A matrix can be reduced by columns in  $O(nN)$  and by rows in  $O(mN)$ . Furthermore a matrix can be reduced completely in  $O(Nmn)$ .*

*Proof.* Compare each pair of columns and delete dominated columns. If the matrix is represented as  $n$  adjacency lists, comparing one column to all others has cost  $O(N)$ , which leads to  $O(nN)$  steps. Note that for advanced reductions the  $f_{\min}(r)$  have to be computed only once. Obviously, the cost for computing the  $f_{\min}(r)$  is as well within  $O(nN)$ . Reduction by rows is analogous. For complete reduction of a matrix, reductions by rows and reductions by columns are applied alternating until no reduction step at all is applicable. In each reduction step at least one row or one column is deleted. Accordingly, the procedure is finished after  $O(\min(m, n))$  alternations. This induces cost  $O(Nmn)$  for complete reduction.  $\square$

*Remark 2.6.* If the matrix is very sparse the analysis can be improved further. Let the number of ones in every row and every column be smaller than a constant  $c$ , then every column has to be compared with at most  $c^2$  other columns, where each comparison has cost  $O(c)$ . Then the matrix is reduced by columns within  $O(c^3n)$  reduction steps. As before it follows that complete reduction can be done in  $O(nm)$ .

*Remark 2.7.* Theorem 2.5 can be slightly improved using results of [9] and [10]: These papers present a lower bound and algorithms for the problem of finding extremal sets, which is equivalent to reduction by column (resp. rows). In [9] a lower bound of  $\Omega(N^2/\log^2(N))$  is proved, and an algorithm with time complexity  $O(N^2/\log(N))$  is given in [10] which is only slightly better than the running time of our much simpler approach.

## 2.5 Instances with Consecutive Ones Property

Instances with C1P can be reduced very effectively. A matrix with C1P maintains this property after every reduction step. If the weights are all equal it can be even reduced to the unit matrix. For the weighted case, reduction can still be done very efficiently by establishing strong C1P and then sorting the rows lexicographically.

*Remark 2.8.* Note that a reduced matrix with C1P for rows has C1P for columns.

SET COVERING for matrices with C1P can be solved by applying a shortest path algorithm in a graph with  $n$  nodes as described in [11]. In our experiments, we used Algorithm 1 which is exponential for general instances but polynomial on instances with C1P.

## 3 A Refined Enumerative Algorithm

For solving a completely reduced set covering problem, it is favorable to apply an algorithm that takes into account the (observed) special structure of the real world instances. More precisely, we aim at an algorithm that is especially effective for instances that are “close” to having C1P. Although the well-established approaches for solving integer linear programs as e.g. branch and bound are very successful in practice, they are often disadvantageous for nicely structured instances.

Alternatively, we designed the following simple, however refined enumerative solution algorithm. The basic idea is simply to enumerate all solutions by combining the covering columns for each row. This leads to partial solutions for all the rows considered so far and finally to a solution for the entire instance. This procedure is described in Algorithm 1.

Throughout Algorithm 1 the cover of partial solutions is maintained in matrix  $B$ , the weights of the columns in array  $w'$ , and the partial solutions in array  $S$ . It always terminates, because in every iteration at least one row is removed from  $C$ . Before each iteration the following invariants hold by induction:

- For every column  $f$  of  $B$ 
  - the partial solution  $S_f$  covers all rows deleted in step 2 up to this iteration and the rows with a one in column  $f$  of  $B$ .
  - $S_f$  has weight  $w'_f$ .
- $S$  contains a subset of an optimal solution for  $I$ .

The key argument is that step 1 adds all solutions covering the first row. The correctness of the returned solution follows if  $C = (1)$ .

In order to keep the effort for computing the partial solutions reasonable, we apply again our reduction technique throughout the algorithm. Of course, the running time of Algorithm 1 is exponential in general. The size of matrix  $B$ , which contains all the partial solutions obtained so far can be still exponentially large. For certain kinds of instances, however, we can prove some nice properties:

**Algorithm 1:** Combi solver**Input** : Instance  $I = (P, F, w, A)$ **Output** : Optimal solution**Start** $B := (0) \in \{0, 1\}^{|P| \times 1}$ , weights  $w' := (0)$ , solutions  $S := (\emptyset)$ ; $C := \left( \begin{array}{c|c} A & B \\ \hline 0 & 1 \end{array} \right)$ ;**while**  $C \neq (1)$  **do**    **forall the** columns  $f$  of  $B$  with 0 in first row **do**        **forall the** columns  $g$  of  $A$  with 1 in first row **do**1             Add a new column  $A_g + B_f$  to  $B$ , a weight  $w_g + w'_f$  to  $w'$  and  
              solution  $S_f \cup \{g\}$  to  $S$ ;        Remove column  $f$ ;2             Remove first row of  $A$  and  $B$  ;              Reduce  $C := \left( \begin{array}{c|c} A & B \\ \hline 0 & 1 \end{array} \right)$  and remove corresponding entries of  $w'$  and  $S$ ;    **return**  $S_1$  and  $w'_1$ ;**End**

**Theorem 3.1.** *If the covering matrix  $A$  has strong C1P, is reduced, the rows are sorted in lexicographic order, and  $c$  is the maximum number of ones per row and per column. Then Algorithm 1 has time complexity  $O(c^3n)$ .*

*Proof.* Let  $c$  be the maximum number of ones in a column or row of  $A$ . If the covering matrix  $A$  has strong C1P, is completely reduced, and the rows are sorted in lexicographic order it has, as noted in remark 2.8, also strong C1P for columns. This implies that after each iteration, the first column of  $B$  contains only zeroes, all other columns have strong C1P, contain a one in the first row, and no two columns are equal. Thus,  $B$  has at most  $c$  columns. So during Algorithm 1 for each column of  $A$  a new column is added to  $B$  at most  $c$  times. Each reduction is in  $O(c^3)$  as stated in Theorem 2.5.  $\square$

Even for a weaker property than C1P Algorithm 1 has polynomial running time:

**Theorem 3.2.** *If the distance between the first and last entry in each column of the covering matrix is at most  $k$ , then at any time throughout Algorithm 1 the number of columns in matrix  $B$  is in  $O(2^k n)$ . The time complexity is in  $O(mn^2 2^{2k} k)$ .*

*Proof.* We prove by induction that after each iteration, the ones in  $B$  are contained in rows 1 to  $k-1$ . As  $B$  is completely reduced, no two columns are equal, so there are at most  $2^k$  different columns. In every iteration only those columns of  $A$  with a one in the first row are considered. The ones of these columns are again contained in rows 1 to  $k$ . Therefore for every column of  $B$  at most  $n$  new columns are added, having all their ones in rows 1 to  $k$ . After addition of all new

columns there are at most  $2^k n$  columns in  $B$  and there are at most  $2^k nk$  ones in  $B$ . So by Theorem 2.5 reduction by columns is in  $O((2^k n)2^k nk)$ . There are at most  $m$  iterations which leads to the claimed upper bound.  $\square$

As Theorem 3.2 indicates, the ordering of the rows is crucial for the running time of the algorithm. We have made good experiences with the Cuthill-McKee ordering (cf. [12]). This means that the rows are chosen in their order of appearance in a breadth first search on the *covering graph* with adjacency matrix  $\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$ . A simple on-line-heuristic which always tries to minimize the size of  $B$  in the next step leads to even better running times.

## 4 Experimental Results

The implementation is in C++ and runs on a i686-Linux platform. The experiments were performed on a Xeon(TM) CPU with 2.80GHz and 2GB RAM.

### 4.1 Instances

**Real World Instances.** We tested our algorithm with different data sets. The original motivation of this work was a project in collaboration with the largest German railway company. Its goal was to optimize the accessibility of customers to local train service. We therefore had access to the data on the German railway network (circa 8200 nodes and 8700 edges) and German settlements (circa 30000). We tested different radii. According to the railway company, radii of two to five kilometers are the most realistic.

We also had two different versions of the problems. The first version contains all settlements that are within the given radius of the railway network, the second version contains only those settlements that were not already covered by one of the (approx. 6800) existing German railway stations. The edges and nodes were weighted by the number of customers traveling through them.

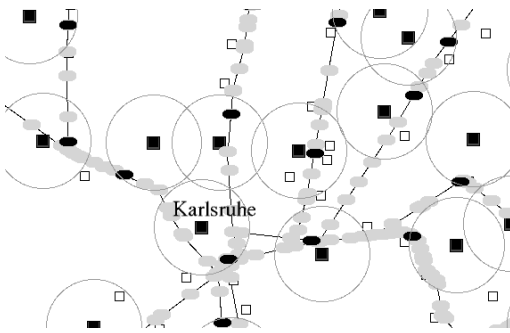
**Generated Instances.** As a second data set we considered randomly generated instances with C1P and suitably modified instances having almost C1P. We experimented with unweighted instances (all weights equal to 1) and instances where the weights were chosen randomly (equally distributed) between 1 and a maximum weight of 10 or 100. The matrices consist of 50 to 50000 rows with 10 to 200 non-zero entries per row. Up to 20% of the ones were randomly flipped to zero resulting in perturbed instances.

### 4.2 Performance of Reduction

**Real World Instances.** The original problem consisted in solving instances that contain only those settlements that are not already covered by existing railway stations. Our experiments on practical data showed that these are always reduced to almost trivial instances in very short time.



Moreover these instances tend to become easier to solve with increasing radius. For large radii many settlements are already covered by old stations and the instance decomposes into many, small components.



The situation is different for instances without existing stations. As an illustrative example for the effect of reduction see the figure on the left. It shows a section of the railway graph near the city of Karlsruhe. The covering radius was 2.5km. The settlements are squares, potential candidates for stops are ellipses. The original instance is colored grey, the instance remaining after reduction is black.

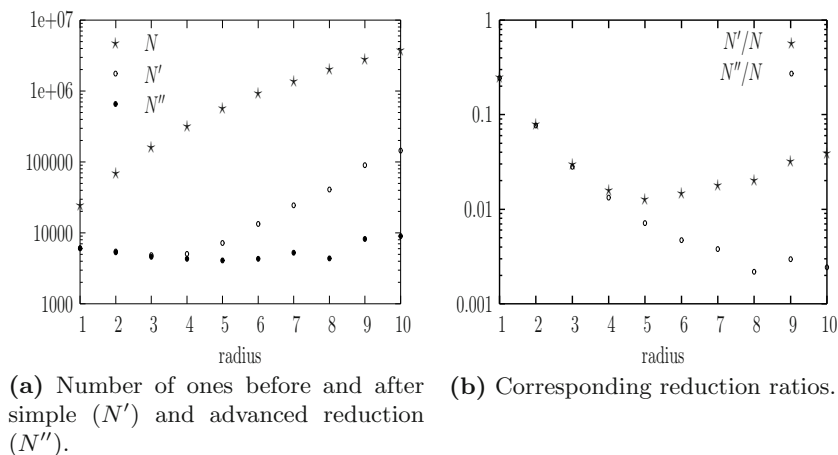
Obviously the solution for the shown part of the remaining instance is already trivial. The largest remaining component of the covering matrix after reduction contained 8 rows and 9 columns.

With increasing radius, however, these instances get quite complex in terms of total matrix size and number and size of independent components. However, reduction is still very effective here. Figure 1 summarizes reduction rates for the German railway graph and radii between 1 and 10km. In these examples the advanced reduction technique is much more effective, especially for large radius. It even seems that the size of the reduced matrix does not grow with increasing radius.

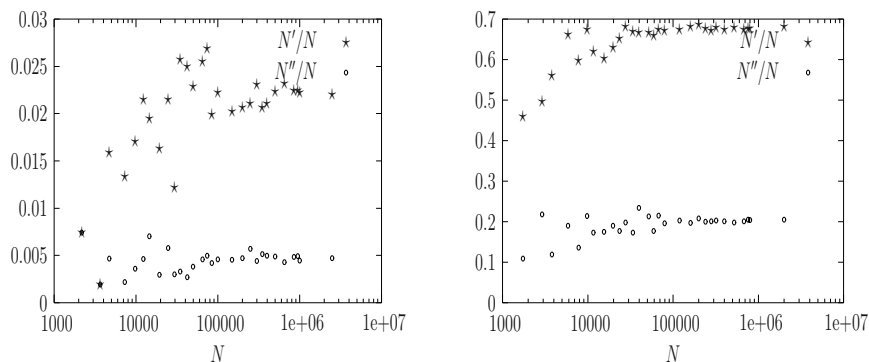
**Generated Instances.** When looking at the C1P matrices, we first observe that the unweighted instances can be reduced completely, that is, to the trivial unit matrix. The weighted matrices could not be reduced completely by the first step, but almost completely by the improved reduction.

Perturbed instances are harder to reduce. For weighted, perturbed instances simple reduction is not very effective but advanced reduction yields very good results as Figure 2 illustrates. Figure 3(a) shows that unweighted instances with 10% perturbation could usually be reduced by a factor of 10, instances with 20% only by a factor of around 2.

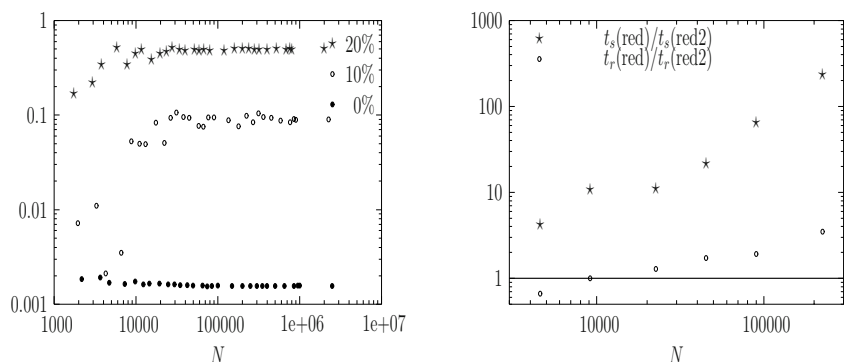
**Increasing the Radius.** Of course,  $N$  grows with increasing radius, but also the combinatorial structure of the instances changes. This affects the effectiveness of reduction. As  $r$  increases the reduction ratio decreases first, increases for moderate radii and finally drops to almost zero for very large radii (The left side of Figure 4 shows this effect for a sub-instance of the german railway graph). For generated instances this effect could only be simulated to a certain extent by just increasing the number of ones per row (see right side of Figure 4).



**Fig. 1.** Effectiveness of reduction for the German railway graph.



**Fig. 2.** Reduction ratios for unperturbed resp. 20% perturbed generated instances.



**Fig. 3(a).** Reduction ratios ( $N'/N$ ), unweighted generated instances.

**Fig. 3(b).** Running time with and without advanced reduction.

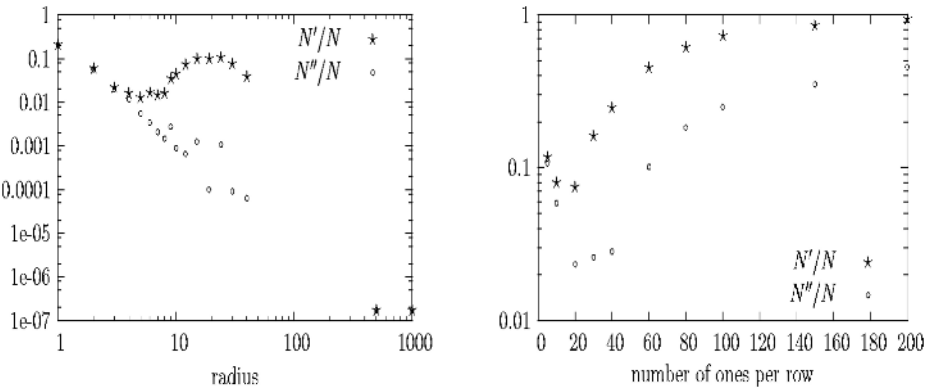


Fig. 4. Reduction ratios for SW-Germany subgraph resp. generated instances.

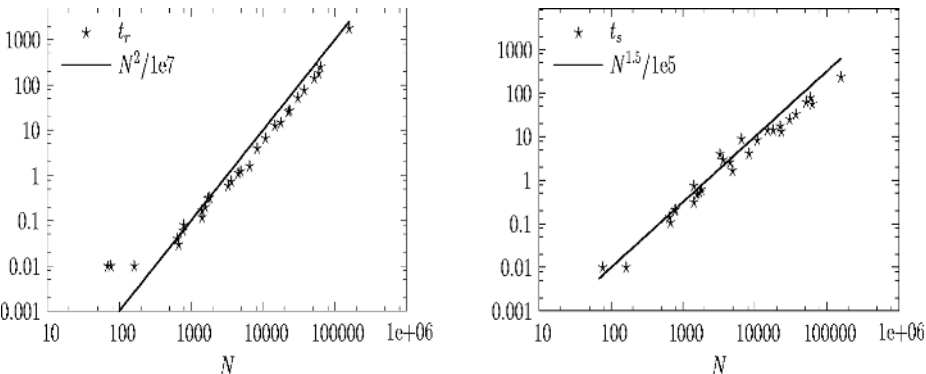


Fig. 5. Reduction time resp. solve time vs. matrix size, weights  $\in (0..10)$ , 10% perturbation.

4.3 Time Complexity

As noted before real world instances with existing stops can be reduced very fast, because they decompose into many, small components. For the variants without existing stations, although the reduction rates were always very good, the running time for solving the instances grew quite fast for radii greater or equal to around 10km. Note, however, that radii larger than 10km are irrelevant in our scenario. For moderate radii we observed a quadratic or slightly subquadratic behaviour (in terms of the matrix size) of the running times for reduction and an almost linear running time for solving the set covering problem.

Looking at the generated instances, our experiments suggest a quadratic time complexity for reduction and an almost linear time complexity for solving the set covering problem (Figure 5). Only for the 20% perturbed, unweighted C1P matrices we observed a quadratic behaviour.

For weighted instances advanced reduction also improved the running time for our solver. Figure 3(b) shows this for the case of 10% perturbed, weighted, generated instances. A speedup factor of 100 and more could be observed here. Even the running time for advanced reduction was usually reduced, owing to a the reduced matrix size after the first rounds of reduction.

## 5 Conclusions

The station location problem was solved almost completely by reduction for our initial real world data. The experimental study confirms the conjecture that one reason for this nice behaviour is the closeness of our instances to C1P. This explains also the results from [7] about the power of data reduction for similar covering problems. Actually it is likely that the data considered there in many cases have (at least almost) C1P.

## References

1. Hamacher, H.W., Liebers, A., Schöbel, A., Wagner, D., Wagner, F.: Locating new stops in a railway network. *Electronic Notes in Theoretical Computer Science*, Proc. ATMOS 2001 **50** (2001)
2. Schöbel, A., Hamacher, H.W., Liebers, A., Wagner, D.: The continuous stop location problem in public transportation networks. Report in *Wirtschaftsmathematik* **81** (2002)
3. Caprara, A., Fischetti, M., Toth, P.: Algorithms for the set covering problem. *Annals of Operations Research* **98** (2000) 353–371
4. Lund, C., Yannakakis, M.: On the hardness of approximating minimization problems. *Journal of the ACM* **41** (1994) 960–981
5. Kranakis, E., Penna, P., Schlude, K., Taylor, D.S., Widmayer, P.: Improving customer proximity to railway stations. In Petreschi, R., Persiano, G., Silvestri, R., eds.: *Proceedings of 5th Italian Conference on Algorithms and Complexity (CIAC 2003)*. Volume 2653 of *Lecture Notes in Computer Science*., Springer (2003) 264–276
6. Nemhauser, G., Wolsey, L.: *Integer and Combinatorial Optimization*. Wiley (1988)
7. Weihe, K.: Covering trains by stations or the power of data reduction. In Battiti, R., Bertossi, A.A., eds.: *Proceedings of Algorithms and Experiments (ALEX 98)*. (1998) 1–8
8. Booth, K.S., Lueker, G.S.: Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Science* **13** (1976) 335–379
9. Yellin, D.M., Jutla, C.S.: Finding extremal sets in less than quadratic time. *Inform. Process. Lett.* **48** (1993) 29–34
10. Pritchard, P.: An old sub-quadratic algorithm for finding extremal sets. *Information Processing Letters* **62** (1997) 329–334
11. Schöbel, A.: *Customer-Oriented Optimization in Public Transportation*. PhD thesis, University of Kaiserslautern (2002) Habilitation.
12. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: *Proc. ACM National Conference, Association of Computing Machinery* (1969)

# Efficient IP Table Lookup via Adaptive Stratified Trees with Selective Reconstructions\*

## (Extended Abstract)

Marco Pellegrini and Giordano Fusco

Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, via Moruzzi  
1, 56124 Pisa, Italy. {marco.pellegrini, giordano.fusco}@iit.cnr.it

**Abstract.** IP address lookup is a critical operation for high bandwidth routers in packet switching networks such as Internet. The lookup is a non-trivial operation since it requires searching for the longest prefix, among those stored in a (large) given table, matching the IP address. Ever increasing routing tables size, traffic volume and links speed demand new and more efficient algorithms. Moreover, the imminent move to IPv6 128-bit addresses will soon require a rethinking of previous technical choices. This article describes a the new data structure for solving the IP table look up problem christened the Adaptive Stratified Tree (AST). The proposed solution is based on casting the problem in geometric terms and on repeated application of efficient local geometric optimization routines. Experiments with this approach have shown that in terms of storage, query time and update time the AST is at a par with state of the art algorithms based on data compression or string manipulations (and often it is better on some of the measured quantities).

## 1 Introduction

**Motivation for the Problem.** Internet is surely one of the great scientific, technological and social successes of the last decade and an ever growing range of services rely on the efficiency of the underlying switching infrastructure. Thus improvements in the throughput of Internet routers are likely to have a large impact. The IP Address Lookup mechanism is a critical component of an Internet Packet Switch (see [1] for an overview). Briefly, a router within the network holds a lookup table with  $n$  entries where each entry specifies a *prefix* (the maximum length  $w$  of a prefix is 32 bits in the IPv4 protocol, 128 bits in the soon-to-be-deployed IPv6 protocol) and a next hop exit line. When a packet comes to the router the destination address in the header of the packet is read, the longest prefix in the table matching the destination is sought, and the packet is sent to the corresponding next hop exit line. How to solve this problem so to be able to handle millions of packets per second has been the topic of a large number of research papers in the last 10-15 years. Steady increase in the size of the lookup tables and relentless demand of traffic performance put pressure on the research community to come up with faster schemes.

---

\* Work partially supported by the Italian Registry of ccTLD .it.

**Table 1.** L2-normalized costs for points of 8 bits. Measures in clock ticks. The first negative entry is due to the inherent approximation in the measurement and normalization of the raw data; as an outlier is not used in the construction of the interpolation formula.

Position	Level				
	1	2	3	4	5
Empty	-3	20	42	66	116
1	12	30	65	93	121
2	12	35	67	94	125
3	16	35	64	100	128
4	22	40	69	101	129

**Our Contribution.** In the IP protocol the address lookup operation is equivalent to the searching for the *longest prefix match* (*lpm*) of a fixed length string amongst a stored set of prefixes. It is well known that the longest prefix match problem can also be mapped into the problem of finding the shortest segment on a line containing a query point. This is called the *geometric* (or *locus*) *view* as opposed to the previous *string view*. The geometric approach has been proposed in [2] and [3]; it has been extended in [4] and in several recent papers [5,6]. It has been used in [7] to prove formally the equivalence among several different problems. Important asymptotic worst case bounds have been found following the geometric point of view, but, currently, the algorithms for IP lookup with the best empirical performance, including the one in [7], are based on the string view and use either optimized tries, or data compression techniques, or both. The question that we rise is whether the geometric view is valuable for the IP lookup problem only as a theoretical tool or it is also a viable tool from a practical point of view. Since our objective is practical we chose for the moment not to rely on the sophisticated techniques that have lead to the recent improvements in worst case bounds (e.g. those in [4,5,6]). Instead we ask ourselves whether simpler search trees built with the help of local optimization routines could lead to the stated goal. The experiments with the AST method lead us to give a strong positive hint on both account: (1) it is indeed possible to gain state of the art performance in IP lookup using the geometric view and (2) local adaptive optimization is a key ingredient for attaining this goal.

**Methodology.** Since this is an experimental paper and the evaluation of the experimental data is critical we stress the experimental methodology beforehand. Performance is established by means of experiments which are described in detail in Sect. 3. The experimental methodology we adopted follows quite closely the standards adopted in the papers describing the methods we compare to. Given the economic relevance of Internet technologies, such state of the art methods have been patented or are in the process of being patented, and, at the best of our knowledge, the corresponding codes (either as source or as executables) are not in the public domain. We refrained from re-implementing these methods, using as guidance the published algorithms, since we are aware that at this level

**Table 2.** List of Input Lookup Tables. Input geometric statistics for active, duplicates and phantom points.

	Routing Table	Date	Entries	Total Points	Active Points	Phantom Points	Duplicate Points
1	Paix	05/30/2001	17,766	35,532	17,807	9,605	8,120
2	AADS	05/30/2001	32,505	65,010	32,642	17,071	15,297
3	Funet	10/30/1997	41,328	82,656	24,895	35,391	22,370
4	Pac Bell	05/30/2001	45,184	90,368	34,334	33,750	22,284
5	Mae West	05/30/2001	71,319	142,638	53,618	50,217	38,803
6	Telstra	03/31/2001	104,096	208,192	64,656	79,429	64,107
7	Oregon	03/31/2001	118,190	236,380	45,299	116,897	74,184
8	AT&T US	07/10/2003	121,613	243,226	102,861	62,910	77,455
9	Telus	07/10/2003	126,282	252,564	85,564	86,738	80,262
10	AT&T East Canada	07/10/2003	127,157	254,314	78,625	93,795	81,894
11	AT&T West Canada	07/10/2003	127,248	254,496	78,708	93,824	81,964
12	Oregon	07/10/2003	142,475	284,950	108,780	84,165	92,005

**Table 3.** Comparisons of IP methods for a 700 MHz Pentium III ( $T = 1.4$  ns) with 1024 KB L2 Cache, L2 delay is  $D = 15$  nsec, L1 latency  $m = 4$  nsec.

Ref	Method	Input size	Formula	Time (ns)	Memory (KB)
[13]	Lulea	38141	$53T + 12D$	254	160
[10]	Variable stride	38816	$26T + 3D$	81	655
[14]	Exp./Con.	44024	$3T + 3D$	49	1057
This paper	Slim AST	45184	$m + 3D$	49	464
This paper	Slim AST	142475	$m + 3D$	49	941

of performance (few dozens of nanoseconds) overlooking seemingly simple implementation details could result in an grossly unfair comparisons. Given the above mentioned difficulties we resolved for an admittedly weaker but still instructive comparison via the fact that those papers present, either an explicit formula for mapping their performance onto different machines, or such formulae could be derived almost mechanically from the description in words. A second issue was deciding the data sets to be used. Papers from the late nineties did use table snapshots usually dating from the mid nineties, when a table of 40,000 entries was considered a large one. Although recovering such old data is probably feasible and using them certainly interesting, the evolution of Internet in recent years has been so rapid that the outcome of comparisons based on outdated test cases is certainly open to the criticism of not being relevant for the present (and the future) of Internet. Therefore the comparisons shown in Table 3, where we use data sets of comparable size and the mapping onto a common architecture given by the formulae, should be considered just in a broad qualitative sense. The main general conclusion we feel we can safely draw is that geometric view is as useful as insight drawn from string manipulation and data compression. We remark that the extensive testing with 12 Lookup Tables of size ranging from

**Table 4.** Slim-AST data structure performance. Tested on a 700 MHz Pentium III ( $T = 1.4$  ns) with 1024 KB L2 Cache, L2 delay is  $D = 15$  nsec, L1 latency  $m = 4$  nsec. The parameters used are:  $B = 16$ ,  $C = 10^{-7}$ ,  $D = 61$  for all tables except (\*) where  $C = 1$ .

	Routing Table	Entries	Target Cache	Total Memory	Worst query (clock-ticks)	Corresponding Class ( $l, p, a$ )
1	Paix	17,766	512	374,314	35	2,2, 8
2	AADS	32,505	512	460,516	35	2,2, 8
3	Funet	41,328	512	453,034	30	2,1, 8
4	Pac Bell	45,184	512	464,063	35	2,2, 8
5	Mae West	71,319	1024	610,587	35	2,2, 8
6	Telstra	104,096	1024	780,452	35	2,2,16
7	Oregon	118,190	1024	595,919	35	2,3, 8
8	AT&T US	121,613	1024	955,702	30	2,1, 8
9	Telus (*)	126,282	1024	878,179	40	2,4, 8
10	AT&T East Canada	127,157	1024	878,155	40	2,4, 8
11	AT&T West Canada	127,248	1024	878,491	40	2,4, 8
12	Oregon (*)	142,475	1024	940,810	35	2,2,16

17,000 to 142,000 (reported in Tables (2), and (4)) confirms the reliability and robustness of the AST.

**Previous Work.** The literature on the IP lookup problem and its higher dimensional extension is rather rich and we refer to the full version for survey. Work most relevant to the AST is mentioned as needed.

**Summary.** The AST is introduced in Sect. 2.1. In Sect. 2.2 the local optimization routines are described. Experiments and results for storage and query time are in Sect. 3, while dynamic operations are discussed in Sect. 4.

## 2 The AST

### 2.1 The AST in a Nutshell

The AST algorithm is described in Section (2.2). Here we summarize the main new ideas. The best way to explain the gist of the algorithm is to visualize it geometrically. We first map the lookup problem into a predecessor search problem using the standard method. The equivalent input for the predecessor problem is a set of labeled points on the real line. We eliminate from the point set duplicates and points for which there is the same label to their left and right (*phantom points*). We want to split this line into a grid of *equal size buckets*, and then proceed recursively and separately on the points contained in each grid bucket. A uniform grid is completely specified by giving an anchor point  $a$  and the step  $s$  of the grid. During the query, finding the bucket containing the query point  $p$  is done easily in time  $O(1)$  by evaluating the expression  $\lfloor (p-a)/s \rfloor$  which



gives the offset from the special bucket containing the anchor. We will take care of choosing for  $s$  a power of two so to reduce integer division to a right shift. If we choose the step  $s$  too short we might end up with too many empty buckets which implies a waste of storage. We choose thus  $s$  as follows: choose the smallest  $s$  for which the ratio of empty to occupied buckets is no more than a user-defined constant threshold. On the other hand shifting the grid (i.e. moving its anchor) can have dramatic effects on the number of empty buckets, occupied buckets and the maximum number of keys in a bucket. So the search for the optimal step size includes an inner optimization loop on the choice of the anchor to minimize the maximum bucket occupation. The construction of the (locally) optimal step and anchor can be done efficiently in time close to linear, up to polylog terms (see Section 2.2). The algorithm works in two main phases, in the first phase we build a tree that aims at using small space at the expense of a few long paths. In the second phase the long paths are shortened by compressing them.

During the construction a basic sub-task is finding a grid optimizing a local measure. This approach has to be contrasted with several techniques in literature where a global optimality criterion is sought usually via much more expensive dynamic programming techniques, such as in [8], [9], [10], and [7]. Ioannidis, Grama and Atallah [11] recently proposed a global reconstruction method for tries that relies on a reduction to the knapsack problem and therefore suggests the use of knapsack approximation methods. In [11] the target is to reduce the average query time based on traffic statistics, not the worst case query time.

A simple theoretical analysis of the asymptotic worst case query time and storage for the AST is shown in [12] giving asymptotic worst case bounds  $O(n)$  for storage and  $O(\log n)$  for query time. However the query time bound does not explain the observed performance and we leave it as an open problem to produce a more refined analysis considering a variety of models.

## 2.2 Construction of the AST

Here we describe the AST data structure by iteratively refining a general framework with specific choices. Consider initially the set  $U = [0, \dots, 2^w - 1]$  of all possible addresses, the set  $P$  of prefixes, and the set  $S = S_P$  of end-points of the segments obtained by mapping each prefix  $b \in P$  to the set of addresses of  $U$  matching  $b$  as described above. We will often think of  $U$  as embedded in the real line  $\mathbb{R}$ . The main idea is to build recursively a tree by levels, each node  $x$  of the tree has an associated connected subset of the universe  $U_x \subset U$ , which is the set of all queries that visit node  $x$ , and a point data set  $S_x \subset S$ , which is  $S \cap U_x$ . For the root  $r$  we have  $U_r = U$  and  $S_r = S$ . Let  $x$  be a node and let  $k_x$  be the number of children on  $x$ . By  $y[1], \dots, y[k_x]$  we denote the children of  $x$ . A particular rule (to be described below) will tell us how to decide  $k_x$  and for each  $i \in [1, \dots, k_x]$  how to compute  $U_{y[i]}$ . Afterwards simply  $S_{y[i]} = U_{y[i]} \cap S_x$ . For each child  $y[i]$ , if  $S_{y[i]}$  is empty we store at  $y[i]$  the next-hop index of the shortest segment covering  $y[i]$ ; and we stop the recursion. For each child  $y[i]$ , if  $S_{y[i]}$  has one point we store at  $y[i]$  the next-hop indices for the two intervals

to the left and right of that point; and we stop the recursion. For simplicity we explain the process at the root. Let  $S$  be a set of points on the real line, and  $\text{span}(S)$  the smallest interval containing  $S$ . Consider an infinite grid  $G(a, s)$  of anchor  $a$  and step  $s$ ,  $G(a, s) = \{a + ks : k \in \mathbb{N}\}$ . By shifting the anchor by  $s$  to the right, the grid remains unchanged:  $G(a, s) = G(a + s, s)$ . Consider the continuous movement of a grid:  $f_G(\alpha) = G(a + \alpha s, s)$  for  $\alpha \in [0, 1]$ . Consider an event a value  $\alpha_i$  for which  $S \cap f_G(\alpha_i) \neq \emptyset$ .

**Lemma 1.** *The number of events is at most  $n$ .*

*Proof.* Take a single fixed interval  $I$  of  $G(a, s)$ , we have an event when the moving left extreme meets a point in  $I$ , this can happen only once for each point in  $I$ . Therefore overall there are at most  $n$  events.  $\square$

Since we study bucket occupancy, extending the shift range beyond 1 is useless since every distribution of points in the bucket has already been considered, given the periodic nature of the grid. Consider point  $p \in S$  and the bucket  $I_p$  containing  $p$ . Point  $p$  produces an event when  $\alpha_p = (p - \text{left}(I_p))/s$  that is when the shift is equal to the distance from the left extreme of the interval containing  $p$ . Thus we can generate the order of events by constructing a min-priority queue  $Q(S, \alpha_p)$  on the set  $S$  using as priority the value of  $\alpha_p$ . We can extract iteratively the minimum for the queue and update the counters  $c(I)$  for the shifting interval  $I$ . Note that for our counters an event consists in decreasing the count for a bucket and increasing it for the neighbour bucket. Moreover we keep the current maximum of the counters. To do so we keep a second max-priority queue  $Q(I, c(I))$ . When a counter increases we apply the operation increase-key, when it decreases we apply the operation decrease-key. Finally we record changes in the root of the priority queue, recording the minimum value found during the life-time of the algorithm. This value is

$$g(s) = \min_{\alpha \in [0, 1]} \max_{I \in G(\alpha s, s)} |S \cap I|$$

that is we find the shift that for a given step  $s$  minimizes the maximal occupancy. The whole algorithm takes time  $O(n \log n)$ . In order to use a binary search scheme we need some monotonicity property which we are going to prove next.

**Lemma 2.** *Take two step values  $s$  and  $t$ , with  $t = 2s$  then we have  $g(t) \geq g(s)$ .*

*Proof.* Consider the grid  $G_{\min}(t)$  that attains min-max occupancy  $K = g(t)$ . So every bucket in  $G(t)$  has at most  $K$  elements. Now we consider the grid  $G(s)$  that splits exactly in two every bucket in  $G_{\min}(t)$ . In this grid  $G(s)$  the maximum occupancy is at most  $K$ , so the value  $g(s)$  that minimizes the maximum occupancy for a translate of  $G(s)$  cannot attain a larger value than  $K$ , i.e.  $g(s) \leq K = g(t)$ .  $\square$

Now if we use only powers of two as possible values for a grid step the above monotonicity lemma implies that, for a given threshold  $x$ , we can use binary search to find the largest step  $s = 2^k$  with  $g(s) \leq x$  and the smallest step  $u = 2^h$  with  $g(u) \geq x$ . Call  $R = E/F$  the ratio of the number empty to the full buckets.

**Lemma 3.** *Take two step values  $s$  and  $t$ , with  $t = 2s$  then we have  $R_{min}(t) \leq R_{min}(s)$ .*

*Proof.* Take the grid  $G_{min}(s)$ , the grid of step  $s$  minimizing the ratio  $R_{min}(s) = E_s/F_s$ . Now make the grid  $G(t)$  by pairing adjacent buckets we have the relations:  $N_s = N$ ,  $N_t = N/2$ ,  $F_s/2 \leq F_t \leq F_s$ , and  $(E_s - F_s)/2 \leq E_t \leq E_s/2$ . Now we express  $R_t = E_t/F_t$  as a function of  $N$  and  $F_t$ .

$$R_t = E_t/F_t = \left(\frac{1}{2}\right) N/F_t - 1.$$

This is an arc of hyperbola (in the variable  $F_t$ ) having maximum value for abscissa  $F_t = F_s/2$ . The value of the maximum is  $E_s/F_s = R_{min}(s)$ . Thus we have shown that  $R(t) \leq R_{min}(s)$ . Naturally also  $R_{min}(t) \leq R(t)$  so we have proved  $R_{min}(t) \leq R_{min}(s)$ . Thus the minimum ratio is monotonic increasing as grids get finer and finer.  $\square$

By the above lemmas we have two functions  $g(\cdot)$  and  $R_{min}(\cdot)$  that are both monotone (one increasing, one decreasing) in the step size that we can compute efficiently.

**Phase I: Saving Space.** In order to control memory consumption we adopt the following criterion recursively and separately at each node: find smallest  $s$  with  $R(s) \leq C$ , for a predefined constant threshold  $C$  then take the grid giving the occupancy  $g(s)$ , the min-max occupancy (this shifting does not change the number of nodes at the level). When the number of points in a bucket drops below a (small) threshold  $2D$  we stop the recursion and we switch to standard sequential search.

**Phase II: Selective Reconstruction.** The tree built in Phase I uses linear space but can have some root-to-leaf path of high length. The purpose of the second phase is to reduce the length of the long paths, without paying too much in storage.

An access the RAM memory is about ten times slower than one to L2 cache. Thus it is fundamental to fit all the relevant data to into the L2 cache. It is reasonable to suppose that the target machine has 1 Mb of L2 cache and that it is completely devoted to the IP Table Lookup. With this hypothesis the size of the routing table does not make difference unless it is below 1 Mb. When the routing table is built, we can perform a selective reconstruction of the deepest paths to flatten the AST in order to reduce the worst query time maintaining the total memory consumption below 1 Mb.

**The cost of a node.** Each node of the AST has associated a cost which represents the time of the slowest query among all the queries that visit that node. Thus the cost of an empty leaf is the time to query a point inside the space of that leaf. The cost of a full leaf is the time to reach the last point stored inside it (i.e. the time to visit all the points from the median to the farthest extreme). The cost of an internal node is the maximum of the costs of all its children.

The leaves can be subdivided into classes  $(l, p, a)$  depending on: their level in the AST, the number of points from the median to the farthest extreme and the number of bits used to store the points in the leaf (8, 16 or 32). To guide the selective reconstruction a Clock-ticks Table is built once for each CPU architecture. For all the possible classes of leaves, the table stores the corresponding query time. The query time of a particular class is measured in the following way: a simple AST containing nodes of the chosen class is built, several queries reaching nodes of that class are performed, the minimum time is taken and it is normalized to L2 cache. The rationale is that a machine dedicated to the IP Table Lookup is capable to perform a query in the minimum time measured in the test machine because the processor is not disturbed by other running processes. The measurements have been done using Pentium specific low level primitives to attain higher precision; nothing forbids to use operating system primitives for better portability.

**Selective Reconstruction.** The following steps are repeated while the size of the Routing Table is below 1 Mb and there are other reconstructions to perform: (i) Create a max-priority queue of internal nodes based on the maximum cost of their children. (ii) Visit the AST and consider only internal nodes having only leaves as children: determine the cost of these nodes and insert them in the max-priority queue. (iii) Extract from the queue the nodes with the same maximum cost and flatten them in the following way: if the maximum number of points in a child full leaf is greater than 1 then split the *step* until the maximum number of points in a bucket becomes smaller than the current maximum; otherwise go the parent and split its *step* until the maximum number of points in a bucket becomes smaller than maximum number of points a full leaf can contain (in this way the level below is eliminated).

A few exceptions to the above rules: the root is not rebuilt, and, for simplicity, when a level is eliminated other reconstructions of same cost are performed only after recomputing the priority queue.

### 3 Experiments with AST

The testing of the (empirical) worst case query time is done statically on the structure of the tree by finding out the longest (most expensive) path from root to leaf. The worst case query time gives valuable information that is relevant for, but independent from, any actual query data stream. Table (2) gives a synthetic view of the IP Lookup Tables used for testing the AST data structure. For each table we report a progressive number, the name of the organization or router holding the table, the date when the Table was downloaded and the number of prefixes in the tables. Each prefix generates two points, next we classify the points into active, phantom and duplicate points, giving the count for each category. Table (4) gives the relevant measure for the Slim-AST. Slim-AST is built only on the active points (without duplicates and phantom points) and it can support

the queries. Updates are supported with the help of an auxiliary data structure that records also the missing data (duplicates and phantoms).

**Normalization in L2.** To test the AST data structure we adopt the methodology in [13]. For completeness we describe the methodology. We visit each node of AST twice so to force data for the second invocation in L1 cache (This is known as the *hot cache* model). We measure number of clock-ticks by reading the appropriate internal CPU registers relative to the second invocation with the schema described above. We also record the number of L1 memory accesses of each query. This is important because, knowing the L1 latency, we decompose the measured time in CPU-operation cost and memory access cost. Afterwards we scale the memory access cost to the cost of accessing data in L2 cache, unless by locality we can argue that the data must be in L1 cache also on the target machine. We call this measurement model the *L2-normalized model*. Since the root is accessed in each query, after unrolling the search loop, we store step, anchor and pointer to the first child at the root in registers so the access to the root is treated separately from accessing any other level of the AST. The results of the L2-normalized measures are shown in Table (1) for 8-bit keys, access times for 16 and 32 bit keys are almost identical.

**Target architecture.** Here we derive a formula we will use to scale our results to other architectures. The target architecture is made of one processor and two caches: one L1 cache and one L2 cache.  $P \leftrightarrow CL1 \leftrightarrow CL2$ . We suppose every memory access is an L1 miss, except when we access consecutive cells. Calling  $M$  the L2 access time,  $m$  the L1 access time, the general formula for accessing a point  $(k, s)$  is:  $A + k(M + m + B) + (M + m) + (s - 1)(C + m)$ . Where we account for any initial constant, for accessing the intermediate nodes, for accessing the leaf and for accessing sequentially points stored at the leaf. Now taking the measurements on our machine and the known values of  $M$  and  $m$  we determine the constants  $A, B, C$ . Since in the slim table we have only results for classes from  $(2, 1)$  to  $(2, 4)$  in Table (1) we need accuracy in this range. Via easy calculations, the final interpolation formula for any  $(2, s)$  class is:  $3(M + m) + (s - 1)m$ . In Table (3) we compare the AST (slim) method with three other methods for which analogous formulae have been derived by the respective authors (explicitly or implicitly), mapped onto a common architecture. The method in [13] is by far the most storage efficient, using roughly 4 bytes per entry; it is based on tree compression techniques, which make updates difficult. The query time is somewhat high. The Expansion/Compression method [14] is very fast (requires only 3 memory access) using a fair amount of memory and requiring extensive reconstructions of the data structure at each update. The method of [10] balances well query time, storage and update time. Its performance for tables of 40,000 entries is not far from the one of the AST. It is not clear however how the balance of resource utilization would scale on tables three times as large. Clearly as processors become faster the time is dominated by the memory access time. So we have our method and [14] converge to roughly the same query time, however our use of storage seems more efficient.

## 4 Dynamic Operations

**Description of the dynamic operations.** We give a brief sketch of the dynamic operations. Inserting a new prefix  $s$  involves two phases: (1) inserting the end-points of the corresponding segment  $M(s)$ ; (2) updating the next-hop index for all the leaves that are covered by the segment  $M(s)$ . While phase (1) involves only two searches and local restructuring, phase (2) involves a DFS of the portion of the AST tree dominated by the inserted/deleted segment. Such a DFS is potentially an expensive operation, requiring in the worst case the visit of the whole tree. However we can trim the search by observing that it is not necessary to visit sub-trees with a span included into a segment that is included in  $M(s)$  since these nodes do not change the next hop. There are  $n$  prefixes and  $O(n)$  leaves, and the span of each leaf is intersected without being enclosed by at most  $2D + 1$  segments, therefore only  $O(1)$  leaves need to be visited on average and eventually updated. Deleting prefix  $s$  from the AST involves phases similar to insertion, however we need to determine first the segment in the AST including  $M(s)$  in order to perform the leaf re-labeling. This is done by performing at most  $w$  searches, one for each prefix of  $s$ , and selecting the longest such prefix present in the AST. Although it is possible to check and enforce the other AST invariants at each update we have noticed that the query performance remains stable over long sequences of random updates. Therefore enforcing the AST invariant is best left to an off-line periodic restructuring process.

**Experimental data.** We perform experiments on updating the AST using as update data stream the full content of each table. For deletions this is natural since we can delete only prefixes present in the table. For insertions, we argue that entries in the table have been inserted in the past, thus it is reasonable to use them as representative also for the future updates. Table (5) shows the number of memory accesses and the percentile time in  $\mu$ -seconds for completing 95%, 99% and 100% of the updates (insertions) for the AST data structures built for the lookup tests. Experiments for deletion give almost identical counts. During the update we count the number of memory cells accessed distinguishing access in L2 and those in L1, assuming the data structure is stored completely in L2 cache. Since in the counting we adopt a conservative policy, i.e. we upper bound the number of cells accessed at each node visited during the update, the final count represents an upper bound on the actual number of memory access. As it is to be expected, the shortest prefixes, corresponding to the longest segments, require a visit of larger portions of the tree and are responsible for the slowest updates.

**Comparison with known methods.** In [10] an estimate of 2500  $\mu$ -sec for worst case updates is given counting the dominant cost of the memory accesses. The cost is incurred when a node in the trie with a large out degree ( $2^{17}$ ) needs to be rebuilt, and an access cost of 20  $nsec$  per entry is assumed. Since 24-bit prefixes are very frequent (about 50% of the entries) the variable stride tries are forced so that updating for such length is faster, requiring roughly 3  $\mu$ -seconds. Although our machine is faster than that used in [10] the L2 access time is quite similar,

**Table 5.** Slim-AST data structure performance of insertion. Number of memory access in L2 and L1 cache for the 100, 99 and 95 percentiles. Time estimate in the test machine. Tested on a 700 MHz Pentium III ( $T = 1.4$  ns) with 1024 KB L2 Cache, L2 delay is  $D = 15$  nsec, L1 latency  $m = 4$  nsec.

	Routing Table	100%			99%			95%		
		Accesses		Time	Accesses		Time	Accesses		Time
		L2	L1	$\mu$ -sec	L2	L1	$\mu$ -sec	L2	L1	$\mu$ -sec
1	Paix	202	26	3.13	59	392	2.45	31	200	1.27
2	AADS	590	8	8.88	59	392	2.45	31	200	1.27
3	Funet	4285	160	64.92	79	492	3.15	61	406	2.54
4	PacBell	1298	156	20.09	71	484	3.00	35	230	1.45
5	MaeWest	1943	86	29.49	89	614	3.79	49	324	2.03
6	Telstra	1948	32	29.35	81	550	3.42	49	326	2.04
7	Oregon	4381	1602	72.12	457	3188	19.61	87	592	3.67
8	AT&T US	1675	80	25.45	69	462	2.88	42	266	1.69
9	Telus	1686	14	25.35	73	492	3.06	45	294	1.85
10	AT&T East Canada	1686	14	25.35	81	544	3.39	49	332	2.06
11	AT&T West Canada	1686	14	25.35	81	544	3.39	49	332	2.06
12	Oregon	2309	80	34.96	467	3246	19.99	66	434	2.73

therefore the timings in [10] can be compared (although only on a qualitative basis) with results in table 5. We believe that the 2500  $\mu$ -sec worst case estimate for [10] is overly pessimistic: in our experiments (deletion and insertion of every entry of every table) we never had to rebuild nodes of very high degree. Updates in [13] and [14] are handled by rebuilding the data structure from scratch, thus requiring time of the order of several milliseconds for every update.

## 5 Conclusions and Open Problems

Preliminary experiments with AST give encouraging results when compared with published results for state of the art methods, however fine tuning of the AST requires choosing properly some user defined parameters and we do not have at the moment general rules for this choice. How to find a theoretical underpinning to explain the AST good empirical performance is still an open issue for further research. A second open line of research is the integration of the AST techniques for empirical efficiency within the framework of the Fat Inverted Segment tree which presently yield good worst case asymptotic bounds.

**Acknowledgments.** We thank Giorgio Vecchiocattivi who contributed to testing an early version of the AST data structure [12], and the participants of the mini-workshop on routing (held in Pisa in March 2003) for many interesting discussions.

## References

1. McKeown, N.: Hot interconnects tutorial slides. Stanford University, available at <http://klamath.stanford.edu/talks/> (1999)
2. Suri, S., Varghese, G., Warkhede, P.: Multiway range trees: Scalable ip lookup with fast updates. Technical Report 99-28, Washington University in St. Luis, Dept. of Computer Science (1999)
3. Lampson, B.W., Srinivasan, V., Varghese, G.: IP lookups using multiway and multicolumn search. *IEEE/ACM Transactions on Networking* **7** (1999) 324–334
4. Feldmann, A., Muthukrishnan, S.: Tradeoffs for packet classification. In: *Proceedings of INFOCOM*, volume 3, IEEE (2000) 1193–1202
5. Thorup, M.: Space efficient dynamic stabbing with fast queries. In: *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, June 9–11, 2003, San Diego, CA, USA, ACM (2003) 649–658
6. Kaplan, H., Molad, E., Tarjan, R.E.: Dynamic rectangular intersection with priorities. In: *Proceedings of the thirty-fifth ACM symposium on Theory of computing*, ACM Press (2003) 639–648
7. Buchsbaum, A.L., Fowler, G.S., Krishnamurthy, B., Vo, K.P., Wang, J.: Fast prefix matching of bounded strings (2003) In *Proceedings of Alenex 2003*.
8. Cheung, G., McCanne, S.: Optimal routing table design for IP address lookups under memory constraints. In: *INFOCOM (3)*. (1999) 1437–1444
9. Gupta, P., Prabhakar, B., Boyd, S.P.: Near optimal routing lookups with bounded worst case performance. In: *INFOCOM (3)*. (2000) 1184–1192
10. Srinivasan, V., Varghese, G.: Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems* (1999) 1–40
11. Ioannidis, I., Grama, A., Atallah, M.: Adaptive data structures for ip lookups. In: *INFOCOM '03*. (2003)
12. Pellegrini, M., Fusco, G., Vecchiocattivi, G.: Adaptive stratified search trees for ip table lookup. Technical Report TR IIT 22/2002, Istituto di Informatica e Telematica del CNR (IIT-CNR), Pisa, Italy (2002)
13. Degermark, M., Brodnik, A., Carlsson, S., Pink, S.: Small forwarding tables for fast routing lookups. In: *SIGCOMM*. (1997) 3–14
14. Crescenzi, P., Dardini, L., Grossi, R.: IP address lookup made fast and simple. In: *European Symposium on Algorithms*. (1999) 65–76



# Super Scalar Sample Sort

Peter Sanders<sup>1</sup> and Sebastian Winkel<sup>2</sup>

<sup>1</sup> Max Planck Institut für Informatik

Saarbrücken, Germany, [sanders@mpi-sb.mpg.de](mailto:sanders@mpi-sb.mpg.de)

<sup>2</sup> Chair for Prog. Lang. and Compiler Construction

Saarland University, Saarbrücken, Germany, [sewi@cs.uni-sb.de](mailto:sewi@cs.uni-sb.de)

**Abstract.** Sample sort, a generalization of quicksort that partitions the input into many pieces, is known as the best practical comparison based sorting algorithm for distributed memory parallel computers. We show that sample sort is also useful on a single processor. The main algorithmic insight is that element comparisons can be decoupled from expensive conditional branching using predicated instructions. This transformation facilitates optimizations like loop unrolling and software pipelining. The final implementation, albeit cache efficient, is limited by a linear number of memory accesses rather than the  $\mathcal{O}(n \log n)$  comparisons. On an Itanium 2 machine, we obtain a speedup of up to 2 over `std::sort` from the GCC STL library, which is known as one of the fastest available quicksort implementations.

## 1 Introduction

Counting comparisons is the most common way to compare the complexity of comparison based sorting algorithms. Indeed, algorithms like quicksort with good sampling strategies [9,14] or merge sort perform close to the lower bound of  $\log n! \approx n \log n$  comparisons<sup>1</sup> for sorting  $n$  elements and are among the best algorithms in practice (e.g., [24,20,5]). At least for numerical keys it is a bit astonishing that comparison operations alone should be good predictors for execution time because comparing numbers is equivalent to subtraction — an operation of negligible cost compared to memory accesses.

Indeed, at least when sorting large elements, it is well known that memory hierarchy effects dominate performance [18]. However, for comparison based sorting of small elements on processors with high memory bandwidth and long pipelines like the Intel Pentium 4, there are almost no visible memory hierarchy effects [5]. Does that mean that comparisons dominate the execution time on these machines? Not quite because execution time divided by  $n \log n$  gives about 8ns in these measurements. But in 8ns the processor can in principle execute around 72 subtractions ( $24 \text{ cycles} \times 3 \text{ instructions per cycle}$ ). To understand what the other 71 slots for instruction execution are (not) doing, we apparently need to have a closer look at processor architecture [8]:

---

<sup>1</sup>  $\log x$  stands for  $\log_2 x$  in this paper.

**Data dependencies** appear when an operand of an instruction  $B$  depends on the result of a previous instruction  $A$ . Such instructions have to be one or more pipeline stages apart.

**Conditional branches** disrupt the instruction stream and impede the extraction of instruction-level parallelism. When a conditional branch enters the first stage of the processor's execution pipeline, its direction must be *predicted* since the actual condition will not be known until the branch reaches one of the later pipeline stages. If a branch turns out to be *mispredicted*, the pipeline must be flushed as it has executed the wrong program path. The resulting penalty is proportional to the pipeline depth. Six cycles are lost on the Itanium 2 and 20 on the Pentium 4 (even 31 on the later 90nm generation). This penalty could be mitigated by speculative execution of both branches, but this causes additional costs and power consumption and quickly becomes hopeless when the next branches come into the way.

Of course, the latter problem is relevant for sorting. To avoid mispredicted branches, much ingenuity has been invested by hardware architects and compiler writers to accurately predict the direction of branches. Unfortunately, all these measures are futile for sorting. For fundamental information theoretic reasons, the branches associated with element comparisons have a close to 50 % chance of going either way if the total number of comparisons is close to  $n \log n$ .

So after discussing a lot of architectural features we are back on square one. Counting comparisons makes sense because comparisons in all known comparison based sorting algorithms are coupled to hard to predict branches. The main motivation for this paper was the question whether this coupling can be avoided, thus leading to more efficient comparison based sorting algorithms (and to a reopened discussion what makes a good practical sorting algorithm).

Section 2 presents super scalar sample sort (sss-sort) — an algorithm where comparisons are *not* coupled to branches. Sample sort [4] is a generalization of quicksort. It uses a random sample to find  $k - 1$  *splitters* that partition the input into *buckets* of about equal size. Buckets are then sorted recursively. Using binary search in the sorted array of splitters, each element is placed into the correct bucket. Sss-sort implements sample sort differently in order to address several aspects of modern architectures:

**Conditional branches** are avoided by placing the splitters into an implicit search tree analogous to the tree structure used for binary heaps. Element placement traverses this tree. It turns out that the only operation that depends on the result of a comparison is an index increment. Such a conditional increment can be compiled into a *predicated instruction*: an instruction that has a predicate register as an additional input and is executed if and only if the boolean value in this register is one. Their simplest form, *conditional moves*, are now available on all modern architectures. If a branch is replaced by these conditionally executed instructions, it can no longer cause mispredictions that disrupt the

pipeline. Loop control overhead is largely eliminated by unrolling the innermost loop of  $\log k$  iterations completely<sup>2</sup>.

**Data dependencies:** Elements can traverse the search tree for placement into buckets independently. By interleaving the corresponding instructions for several elements, data dependencies cannot delay the computation. Using a two-pass approach, sss-sort makes it easy for the compiler to implement this optimization (almost) automatically using further loop unrolling or software pipelining. In the first pass, only an “oracle” — the correct bucket number — is remembered and the bucket sizes are counted. In the second pass, the oracles and the bucket sizes are used to efficiently place the elements into preallocated subarrays. This two-pass approach is well known from algorithms based on radix sort, but only the introduction of oracles make it feasible for comparison based sorting. The two-pass approach has the interesting side effect that the comparisons and the most memory intensive components of the algorithm are cleanly separated and we can easily find out what dominates execution time.

**Memory Hierarchies:** Sample sort is more cache efficient than quicksort because it moves the elements only  $\log_k n$  times rather than  $\log n$  times. The parameter  $k$  depends on several properties of the machine.

The term “super scalar” in the algorithm name, as in [1], says that this algorithm enables the use of instruction parallelism by getting rid of hard to predict branches and data dependencies. Were it not for the alliteration, “instruction parallel sample sort” might have been a better name since also superpipelined, VLIW, and explicitly parallel instruction architectures can profit from sss-sort.

Section 3 gives results of an implementation of sss-sort on Intel Itanium 2 and Pentium 4 processors. The discussion in Section 4 summarizes the results and outlines further refinements.

## Related Work

So much work has been published on sorting and so much folklore knowledge is important that it is difficult to give an accurate history of what is considered the “best” practical sorting algorithm. But it is fair to say that refined versions of Hoare’s quicksort [9,17] are still the best general purpose algorithms: Being comparison based, quicksort makes little assumptions on the key type. Choosing splitters based on random samples of size  $\Theta(\sqrt{n})$  [14] gives an algorithm using only  $n \log n + \mathcal{O}(n)$  expected comparisons. A proper implementation takes only  $\mathcal{O}(\log n)$  additional space, and a fallback to heapsort ensures worst case efficiency. In a recent study [5], `STL::sort` from the GCC STL library fares best among all quicksort competitors. This routine stems from the HP/SGI STL library and implements *introsort*, a quicksort variant described in [17]. Only for large  $n$  and only on some architectures this implementation is outperformed by more cache efficient algorithms. Since quicksort only scans an array from both ends, it has perfect spatial locality even for the smallest of caches. Its only disadvantage is

<sup>2</sup> Throughout this paper, we assume that  $k$  is a power of two.

**Table 1.** Different complexity measures for  $k$ -way distribution and  $k$ -way merging in comparison based sorting algorithms. All these algorithms need  $n \log k$  element comparisons. Lower order terms are omitted. Branches: number of hard to predict branches; data dep.: number of instructions that depend on another close-by instruction; I/Os: number of cache faults assuming the I/O model with block size  $B$ ; instructions: necessary size of instruction cache.

	mem. acc.	branches	data dep.	I/Os	registers	instructions
$k$ -way distribution:						
sss-sort	$n \log k$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$3.5n/B$	$3 \times \text{unroll}$	$\mathcal{O}(\log k)$
quicksort $\log k$ lvls.	$2n \log k$	$n \log k$	$\mathcal{O}(n \log k)$	$2 \frac{n}{B} \log k$	4	$\mathcal{O}(1)$
$k$ -way merging:						
memory [22,12]	$n \log k$	$n \log k$	$\mathcal{O}(n \log k)$	$2n/B$	7	$\mathcal{O}(\log k)$
register [24,20]	$2n$	$n \log k$	$\mathcal{O}(n \log k)$	$2n/B$	$k$	$\mathcal{O}(k)$
funnel $k'^{\log_{k'} k}$ [5]	$2n \log_{k'} k$	$n \log k$	$\mathcal{O}(n \log k)$	$2n/B$	$2k' + 2$	$\mathcal{O}(k')$

that it reads and writes the entire input  $\log \frac{n}{M}$  times before the subproblems fit into a cache of size  $M$ .

Recent work on engineering sequential sorting algorithms has focused on more cache efficient algorithms based on divide-and-conquer that split the problem into subproblems of size about  $n/k$  and hence only need to read and write the input  $\log_k \frac{n}{M}$  times. In the I/O model [2], with an omniscient cache of size  $M$  and accesses (“I/Os”) of size  $B$ , we have  $k = \mathcal{O}(M/B)$ . In practical implementations,  $k$  has to be reduced by a factor  $\Omega(B^{1/a})$  for  $a$ -way associative hardware caches [15] (this restriction can in principle be circumvented [23]). On current machines, the size of the Translation Lookaside Buffer (TLB) is the more stringent restriction. This translation table keeps the physical address of the most recently accessed pages of virtual memory. Access to other pages causes *TLB misses*. TLBs are small (usually between 64 and 256) and for large  $n$ , TLB misses can get much more expensive than cache misses.

*k*-way Merging is a simple deterministic approach to  $k$ -way divide-and-conquer sorting and is the most popular approach to cache efficient comparison based sorting [18,13,22,24,20,5]. Using a *tournament tree* data structure [12], the smallest remaining element from  $k$  sorted data streams can be selected using  $\log k$  comparisons. Keeping this tournament tree in registers reduces the number of memory accesses by a factor  $\log k$  at the cost of  $2k$  registers and a cumbersome case distinction in the inner loop [22,20,24]. Although register based multi-way merging severely restricts the maximal  $k$ , this approach can be a good compromise for many architectures [20,24]. This is another indication that most sorting algorithms are less restricted by the memory hierarchy than by data dependencies and delays for branches. Register based  $k'$ -way mergers can be arranged into a cache efficient  $k$ -way *funnel merger* by coupling  $\log \frac{k}{k'}$  levels using buffer arrays [5]. By appropriately choosing the sizes of the buffer arrays, this algorithm becomes *cache-oblivious*, i.e., it works without knowing the I/O-model parameters  $M$  and  $B$  and is thus efficient on all levels of the memory hierarchy [7].

**Function** *sampleSort*( $e = \langle e_1, \dots, e_n \rangle, k$ )  
  **if**  $n/k$  is “small” **then return** *smallSort*( $e$ )                   // base case, e.g. quicksort  
  **let**  $\langle S_1, \dots, S_{ak-1} \rangle$  **denote a random sample of**  $e$   
  **sort**  $S$                    // or at least locate the elements whose rank is a multiple of  $a$   
   $\langle s_0, s_1, s_2, \dots, s_{k-1}, s_k \rangle := \langle -\infty, S_a, S_{2a}, \dots, S_{(k-1)a}, \infty \rangle$    // determine splitters  
  **for**  $i := 1$  **to**  $n$  **do**  
    **find**  $j \in \{1, \dots, k\}$  **such that**  $s_{j-1} < e_i \leq s_j$   
    **place**  $e_i$  **in bucket**  $b_j$   
  **return** *concatenate*(*sampleSort*( $b_1$ ),  $\dots$ , *sampleSort*( $b_k$ ))

**Fig. 1.** Pseudocode for sample sort with  $k$ -way partitioning and oversampling factor  $a$ .

For integer keys, radix sort with  $\log k$  bit digits starting with the most significant digit (MSD) can be very fast. This is probably folklore. We single out [1] because it contains the first mention of TLB as an important factor in choosing  $k$  and because it starts with the sentence “The compare and branch sequences required in a traditional sort algorithm cannot efficiently exploit multiple execution units present in currently available RISC processors.” However, radix sort in general and this implementation in particular depend heavily on the length and distribution of input keys. In contrast, sss-sort does away with compare and branch sequences without assumptions on the input keys. For a recent overview of radix sort implementations refer to [19,11]. Sss-sort owes a lot to MSD radix sort because it is also distribution based and since it adopts the two-pass approach of a counting phase followed by a distribution phase.

One goal of this paper is to give an example of how it can be algorithmically interesting and relevant for performance to consider complexity measures beyond the RAM model and memory hierarchies. Table 1 summarizes the complexity of  $k$ -way distribution and  $k$ -way merging for five different algorithms and five different complexity measures (six if we count comparisons). For the sake of this comparison,  $\log k$  recursion levels of quicksort are viewed as a means of  $k$ -way distribution. We can see that sss-sort outperforms the other algorithms with respect to the conditional branches and data dependencies. It also fares very well regarding the other measures. The constant factor in the number of I/Os might be improvable. Refer to Section 4 for a short discussion.

## 2 Super Scalar Sample Sort

Our starting point is ordinary sample sort. Fig. 1 gives high level pseudocode. Small inputs are sorted using some other algorithm like quicksort. For larger inputs, we first take a sample of  $s = ak$  randomly chosen elements. The *oversampling factor*  $a$  allows a flexible tradeoff between the overhead for handling the sample and the accuracy of splitting. In the full paper we give a heuristic derivation of a good oversampling factor. Our splitters are those elements whose rank in the sample is a multiple of  $a$ . Now each input element is located in

the splitters and placed into the corresponding bucket. The buckets are sorted recursively and their concatenation is the sorted output.

Sss-sort is an implementation strategy for the basic sample sort algorithm. We describe one simple variant here and refer to Section 4 for a discussion of some possible refinements. All sequences are represented as arrays. More precisely, we need two arrays of size  $n$ . One for the original input and one for temporary storage. The flow of data between these two arrays alternates in different levels of recursion. If the number of recursion levels is odd, a final copy operation makes sure that the output is in the same place as the input. Using an array of size  $n$  to accommodate all buckets means that we need to know exactly how big each bucket is. In radix sort implementations this is done by locating each element twice. But this would be prohibitive in a comparison based algorithm. Therefore we use an additional auxiliary array,  $o$ , of  $n$  oracles –  $o(i)$  stores the bucket index for  $e_i$ . A first pass computes the oracles and the bucket sizes. A second pass reads the elements again and places element  $e_i$  into bucket  $b_{o(i)}$ . This two pass approach incurs costs in space and time. However these costs are rather small since bytes suffice for the oracles and the additional memory accesses are sequential and thus can almost completely be hidden via software or hardware prefetching [8,10]. In exchange we get simplified memory management, no need to test for bucket overflows. Perhaps more importantly, decoupling the expensive tasks of finding buckets and distributing elements to buckets facilitates software pipelining [3] by the compiler and prevents cache interferences of the two parts. This optimization is also known as *loop distribution* [16,6].

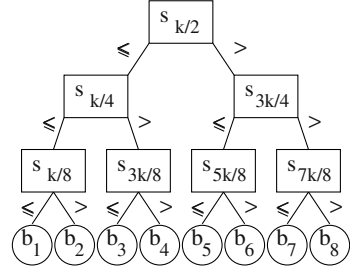
Theoretically the most expensive and algorithmically the most interesting part is how to locate elements with respect to the splitters. Fig. 2 gives pseudocode and a picture for this part. Assume  $k$  is a power of two. The splitters are placed into an array  $t$  such that they form a complete binary search tree with root  $t_1 = s_{k/2}$ . The left successor of  $t_j$  is stored at  $t_{2j}$  and the right successor is stored at  $t_{2j+1}$ . This is the arrangement well known from binary heaps but used for representing a search tree here. To locate an element  $a_i$ , it suffices to travel down this tree, multiplying the index  $j$  by two in each level and adding one if the element is larger than the current splitter. This increment is the only instruction that depends on the outcome of the comparison. Some architectures like IA-64 have predicated arithmetic instructions that are only executed if the previously computed condition code in the instruction's predicate register is set. Others at least have a conditional move so that we can compute  $j' := 2j$  and then, speculatively,  $j' := j + 1$ . Then we conditionally move  $j'$  to  $j$ . The difference between such predicated instructions and ordinary branches is that they do not affect the instruction flow and hence cannot suffer from branch mispredictions.

When the search has traveled down to the bottom of the tree, the index  $j$  lies between  $k$  and  $2k - 1$  so that subtracting  $k - 1$  yields the bucket index  $o(i)$ . Note that each iteration of the inner loop needs only four or five machine instructions so that when unrolling it completely, even for  $\log k = 8$  we still have only a small number of instructions. We also need only three registers for maintaining the state of the search ( $a_i$ ,  $t_j$ , and  $j$ ). Since modern processors have many more

```

 $t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \dots \rangle$  //
for  $i := 1$  to  $n$  do // locate each element
   $j := 1$  // current tree node := root
  repeat  $\log k$  times // will be unrolled
     $j := 2j + (a_i > t_j)$  // left or right?
   $j := j - k + 1$  // bucket index
   $|b_j|++$  // count bucket size
   $o(i) := j$  // remember oracle

```



**Fig. 2.** Finding buckets using implicit search trees. The picture is for  $k = 8$ . We adopt the C convention that “ $x > y$ ” is one if  $x > y$  holds, and zero else.

physical renaming registers<sup>3</sup>, we can afford to run several iterations of the outer loop in an interleaved fashion (via unrolling or *software pipelining*) in order to overlap their execution.

With the oracles it is very easy to distribute the input elements in array  $a$  to an output array  $a'$  that stores all the buckets consecutively and without any gaps. Assume that  $B[j]$  is initialized to  $\sum_{i < j} |b_i|$ . Then the distribution is a one-liner:

```

for  $i := 1$  to  $n$  do  $a'_{B[o(i)]++} := a_i$  // (*)

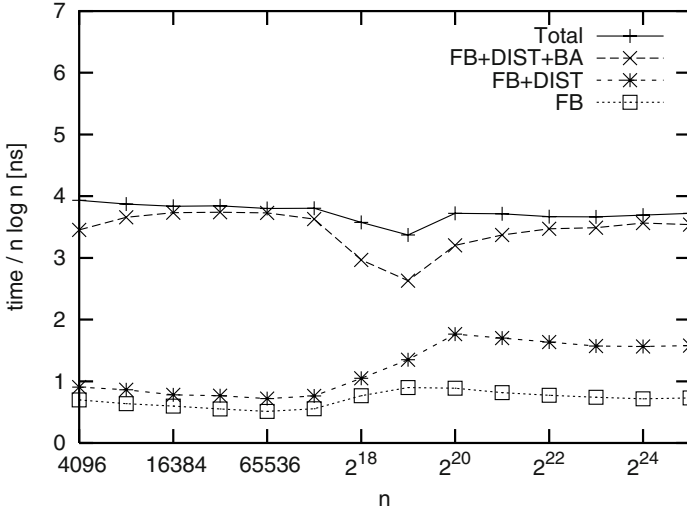
```

This is quite fast in practice because  $a$  and  $o$  are read linearly, which enables prefetching. Writing to  $a'$  happens only on  $k$  distinct positions so that caching is effective if  $k$  is not too big and if all currently accessed pages fit into the TLB. Data dependencies only show up for the accesses to  $B$ . But this array is very small and hence fits in the first-level cache.

### 3 Experiments

We have implemented sss-sort on an Intel server running Red Hat Enterprise Linux AS 2.1 with four 1.4 GHz Itanium 2 processors and 1 GByte of RAM. We used Intel’s C++ compiler v8.0 (Build 20031017) [6]. We have chosen this platform because the hardware and, more importantly, the compiler have good support for the two main architectural features we exploit: predicated instructions and software pipelining. We would like to stress, however, that the simple type of predicated instructions we need – the conditional moves – are supported by all modern processor architectures like the Intel Pentium II/III/4, the AMD Athlon and Opteron, Alpha, PowerPC and the IBM Power3/4/5 processors. Hence, with an appropriate compiler or manual coding, our algorithm should

<sup>3</sup> For instance, the Pentium 4 has 126 physical registers. The eight architected registers of IA-32 are internally mapped to these physical registers via renaming, allowing to resolve all false dependencies which are due to the limited number of architected registers [8]. In our case, this means that different iterations of the outer loop can be mapped to different registers, allowing them to be executed in parallel.



**Fig. 3.** Breakdown of the execution time of sss-sort (divided by  $n \log n$ ) into phases. “FB” denotes the finding of buckets for the elements, “DIST” the distribution of the elements to the buckets, “BA” the base sorting routines. The remaining time is spent in finding the splitters etc.

work well on most machines (like on the Pentium 4, as demonstrated at the end of this section).

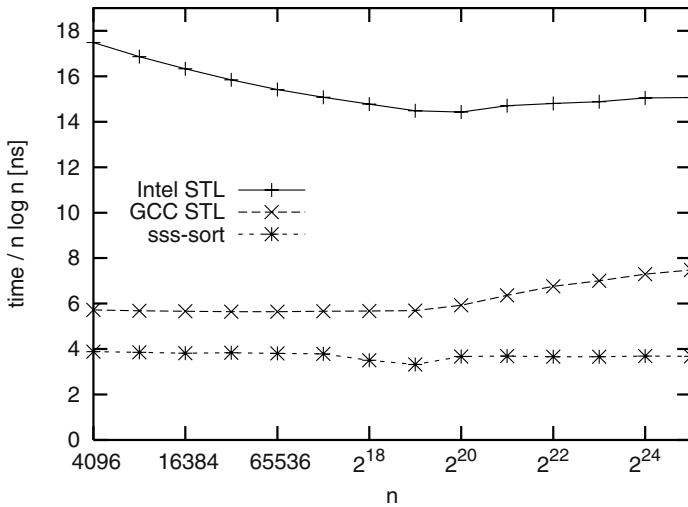
We have applied the `restrict` keyword from the ANSI/ISO C standard C99 several times to communicate to the compiler that our different arrays are not overlapped and not accessed through any other aliased pointer. This was necessary to enable the compiler to software pipeline the two major critical loops (“find the buckets” from Fig. 2, abbreviated “FB” in the following, and “distribute to buckets” ( $*$ ) in Sec. 2), abbr. “DIST”).

Prefetch instructions and predication are utilized automatically by the compiler as intended in Section 2. The used compiler flags are “-O3 -prof.use -restrict”, where “-prof.use” indicates that we have performed profiling runs (separately for each tested sorting algorithm). Profiling gives a 9% speedup for sss-sort. Our implementation uses  $k = 256$  to allow byte-size oracles. sss-sort calls itself recursively to sort buckets of size larger than 1000 (cf. Fig. 1). It uses quicksort between 100 and 1000, insertion sort between 5 and 100, and customized straight-line code for  $n \leq 5$ .

Below we report on experiments for 32 bit random integers in the range  $[0, 10^9]$ . We have not varied the input distribution since sample sort using random sampling is largely independent of the input distribution.<sup>4</sup> The figures have the execution time divided by  $n \log n$  for the  $y$  axis, i.e., we give something like “time

<sup>4</sup> Our current simple implementation would suffer in presence of many identical keys. However, this could be fixed without much overhead: If  $s_{i-1} < s_i = s_{i+1} = \dots = s_j$ ,  $j > i$ , change  $s_i$  to  $s_i - 1$ . Do not recurse on buckets  $b_{i+1}, \dots, b_j$  – they all contain identical keys.



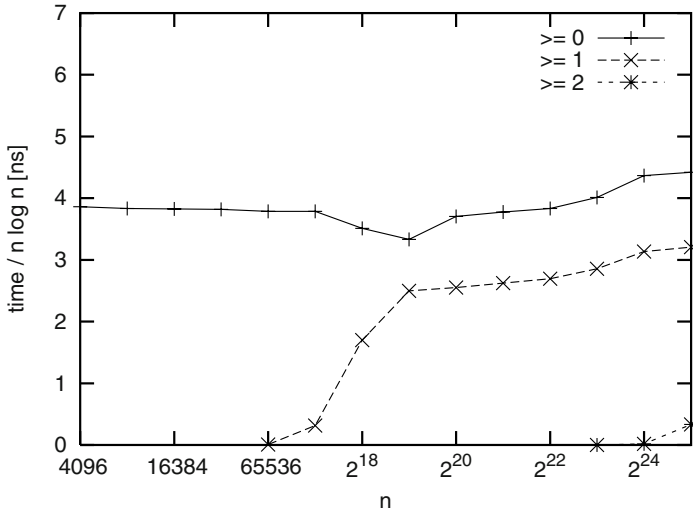


**Fig. 4.** Execution time on the Itanium, divided by  $n \log n$ , for the Intel implementation of `STL::sort`, the corresponding GCC implementation and for `sss-sort`. The measurements were repeated sufficiently often to require at least 10 seconds for all algorithms.

per comparison”. For a  $\Theta(n \log n)$  algorithm one would expect a flat line in the traditional RAM model of computation. Deviations from this expectations are thus signals for architectural effects.

Fig. 3 provides a breakdown of the execution time of `sss-sort` into different phases. It is remarkable that the central phases `FB` and `DIST` account only for a minor proportion of the total execution time. The reason is that software pipelining turns out to be highly effective here: It reduces the schedule lengths of the loops `FB` and `DIST` from 60 and 6 to 11 and 3 cycles, respectively (this can be seen from the assembly code on this statically scheduled architecture). Furthermore, we have measured that for relatively small inputs ( $n \leq 2^{18}$ ), it takes on the average only 11.7 cycles to execute the 63 instructions in the loop body of `FB`, and 4 cycles to execute the 14 instructions in that of `DIST`. This yields dynamic IPC (instructions per clock) rates of 5.4 and 3.5, respectively; the former is not far from the maximum of 6 on this architecture.

The IPC rate of `FB` decreases only slightly as  $n$  grows and is still high with 4.5 at  $n = 2^{25}$ . In contrast, the IPC rate of `DIST` begins to decrease when the memory footprint of the algorithm (approx.  $4n+4n+n=9n$  bytes) approaches the L3 cache size of 4 MB (this can also be seen in Fig. 3, starting at  $n = 2^{17}$ ). Then the stores in `DIST` write through the caches to the main memory and experience a significant latency, but on the Itanium 2, not more than 54 store requests can be queued throughout the memory hierarchy to cover this latency [10,21]: Consequently, the stall times during the execution of `DIST` increase (also those due to TLB misses). The IPC of its loop body drops to 0.8 at  $n = 2^{25}$ .



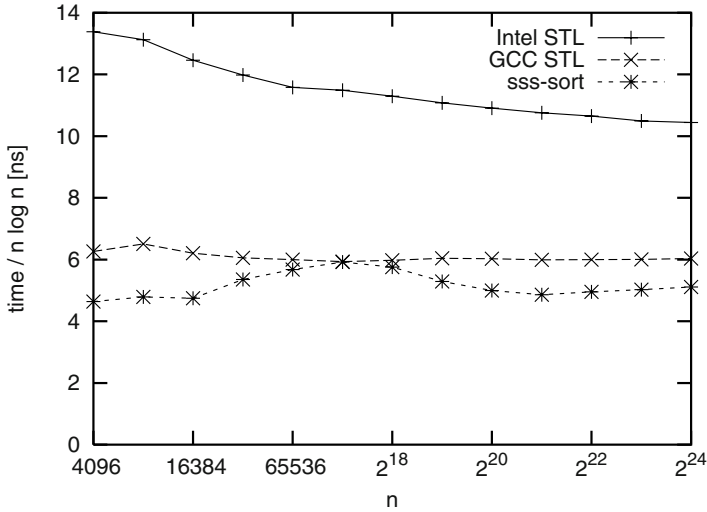
**Fig. 5.** The execution time spent in the different recursion levels of sss-sort. The total execution time ( $\geq 0$ ) shows small distortions compared to Fig. 3, due to the measurement overhead.

However, Fig. 3 also shows — together with Fig. 5 — that at the same time the first recursion level sets in and reduces the time needed for the base cases, thus ameliorating this increase.

Fig. 4 compares the timing for our algorithm with two previous quicksort implementations. The first one is `std::sort` from the Dinkumware STL library delivered with Intel’s C++ compiler v8.0. The second one is `std::sort` from the STL library included in GCC v3.3.2. The latter routine is much faster than the Dinkumware implementation and fared extremely well in a recent comparison of the best sorting algorithms [5]: There it remained unbeaten on the Pentium 4 and was outperformed on the Itanium 2 only by funnelsort for  $n \geq 2^{24}$  (by a factor of up to 1.18).

As Fig. 4 shows, our sss-sort beats its closest competitor, GCC STL, by at least one third; the gain over GCC STL grows with  $n$  and reaches more than 100%. The flatness of the sss-sort curve in the figure (compared to GCC STL) demonstrates the high cache efficiency of our algorithm.

We have repeated these experiments on a 2.66 GHz Pentium 4 (512 KB L2 cache) machine running NetBSD 1.6.2, using the same compiler in the IA-32 version (Build 20031016). Since the latter does not perform software pipelining, we have unrolled each of the loops FB and DIST twice to expose the available parallelism. We have added the flag “-xN” to enable optimizations for the Pentium 4, especially the conditional moves. These were applied in FB as intended, except for four move instructions, which were still dependent on conditional branches. These moves were turned into conditional moves manually in order to obtain a branchless loop body.



**Fig. 6.** The same comparison as in Figure 4 on a Pentium 4 processor.

The breakdown of the execution time, available in the full paper, shows that FB and DIST are here more expensive than on the Itanium. One reason for this is the lower parallelism of the Pentium 4 (the maximal sustainable IPC is about 2; we measure 1.1-1.8 IPC for the loop body of FB). Nevertheless, sss-sort manages to outperform GCC STL for most input sizes by 15-20% (Figure 6). The drop at  $n = 2^{17}$  could probably be alleviated by adjusting the splitter number  $k - 1$  and the base sorting routines.

## 4 Discussion

Sss-sort is a comparison based sorting algorithm whose  $\mathcal{O}(n \log n)$  term contains only very simple operations without unnecessary conditional branches or short range data dependencies on the critical path. The result is that the cost for this part on modern processors is dwarfed by “linear” terms. This observation underlines that algorithm performance engineering should pay more attention to technological properties of machines like parallelism, data dependencies, and memory hierarchies. We have to take these aspects into account even if they show up in lower order terms that contribute a logarithmic factor less instructions than the “inner loop”.

Apart from these theoretical considerations, sss-sort is a candidate for being the best comparison based sorting algorithm for large data sets. Having made this claim, it is natural to discuss remaining weaknesses: So far we have found only one architecture with a sufficiently sophisticated compiler to fully harvest the potential advantages of sss-sort. Perhaps sss-sort can serve as a motivating example for compiler writers to have a closer look at exploiting predication.

Another disadvantage compared to quicksort is that sss-sort is not inplace. One could make it almost inplace however. This is most easy to explain for the case that both input and output are a sequence of blocks, holding  $c$  elements each for some appropriate parameter  $c$ . The required pointers cost space  $\mathcal{O}(n/c)$ . Sampling takes sublinear space and time. Distribution needs at most  $2k$  additional blocks and can otherwise recycle freed blocks of the input sequence. Although software pipelining may be more difficult for this distribution loop, the block representation facilitates a single pass implementation without the time and space overhead for oracles so that good performance may be possible. Since it is possible to convert inplace between block list representation and an array representation in linear time, one could actually attempt an almost inplace implementation of sss-sort.<sup>5</sup> If  $c$  is sufficiently big, the conversion should not be much more expensive than copying the entire input once, i.e., we are talking about conversion speeds of GBytes per second.

Algorithms that are conspicuously absent from Tab. 1 are distribution based counterparts of register based merging and funnel-merging. Register based distribution, i.e., keeping the search tree in registers would be possible but it would reintroduce expensive conditional branches and only gain some rather cheap in-cache memory accesses. On the other hand, cache-oblivious or multi-level cache-aware  $k$ -way distribution might be interesting (see also [7]). An interesting feature of sss-sort in this respect is that the two-phase approach allows us to precompute the distribution information for a rather large  $k$  (in particular larger than the TLB) and then implement the actual distribution exploiting several levels of the memory hierarchy. Let us consider an example. Assume we have  $2^{23}$  bytes of L3 cache and want to distribute with  $k = 2^{14}$ . Then we distribute  $2^{23}$  byte batches of data to a temporary array. This array will be in cache and its page addresses will be in the TLB. After a batch is distributed, each bucket is moved to its final destination in main memory. The cost for TLB misses will now be amortized over an average bucket size of  $2^9$  bytes. A useful side effect is that cache misses due to the limited associativity of hardware caches are eliminated [23,15].

**Acknowledgments.** This work was initiated at the Schloss Dagstuhl Perspectives Workshop "Software Optimization" organized by Susan L. Graham and Reinhard Wilhelm. We would also like to thank the HP TestDrive team for providing access to Itanium systems.

## References

1. R. Agarwal. A super scalar sort algorithm for RISC processors. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 240–246, 1996.
2. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

---

<sup>5</sup> More details are provided in a manuscript in preparation.

3. V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software Pipelining. *Computing Surveys*, 27(3):367–432, September 1995.
4. G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 3–16, 1991.
5. G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In *6th Workshop on Algorithm Engineering and Experiments*, 2004.
6. Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John Ng, and David Sehr. An Overview of the Intel® IA-64 Compiler. *Intel Technology Journal*, (Q4), 1999.
7. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Symposium on Foundations of Computer Science*, pages 285–298, 1999.
8. J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2002.
9. C. A. R. Hoare. Quicksort. *Communication of the ACM*, 4(7):321, 1961.
10. Intel. *Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization*, April 2003.
11. D. Jiminez-Gonzalez, J.-L. Larriba-Pey, and J. J. Navarro. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, chapter Case Study: Memory Conscious Parallel Sorting, pages 171–192. Springer, 2003.
12. D. E. Knuth. *The Art of Computer Programming — Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.
13. A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *8th Symposium on Discrete Algorithm*, pages 370–379, 1997.
14. C. Martínez and S. Roura. Optimal sampling strategies in Quicksort and Quickslect. *SIAM Journal on Computing*, 31(3):683–705, June 2002.
15. K. Mehlhorn and P. Sanders. Scanning multiple sequences via cache memory. *Algorithmica*, 35(1):75–93, 2003.
16. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, Kalifornien, 1997.
17. David R. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exper.*, 27(8):983–993, 1997.
18. C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC machine sort. In *SIGMOD*, pages 233–242, 1994.
19. N. Rahman. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, chapter Algorithms for Hardware Caches and TLB, pages 171–192. Springer, 2003.
20. A. Ranade, S. Kothari, and R. Udupa. Register efficient mergesorting. In *High Performance Computing — HiPC*, volume 1970 of *LNCS*, pages 96–103. Springer, 2000.
21. Reid Riedlinger and Tom Grutkowski. The High Bandwidth, 256KB 2nd Level Cache on an Itanium™ Microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, San Francisco, February 2002.
22. Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.
23. S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *11th ACM Symposium of Discrete Algorithms*, pages 829–838, 2000.
24. R. Wickremesinghe, L. Arge, J. S. Chase, and J. S. Vitter. Efficient sorting using registers and caches. *ACM Journal of Experimental Algorithmics*, 7(9), 2002.

# Construction of Minimum-Weight Spanners

Mikkel Sigurd and Martin Zachariasen

Department of Computer Science, University of Copenhagen  
DK-2100 Copenhagen Ø, Denmark  
{sigurd,martinz}@diku.dk.

**Abstract.** Spanners are sparse subgraphs that preserve distances up to a given factor in the underlying graph. Recently spanners have found important practical applications in metric space searching and message distribution in networks. These applications use some variant of the so-called *greedy* algorithm for constructing the spanner — an algorithm that mimics Kruskal’s minimum spanning tree algorithm. Greedy spanners have nice theoretical properties, but their practical performance with respect to total weight is unknown. In this paper we give an exact algorithm for constructing minimum-weight spanners in arbitrary graphs. By using the solutions (and lower bounds) from this algorithm, we experimentally evaluate the performance of the greedy algorithm for a set of realistic problem instances.

## 1 Introduction

Let  $G = (V, E)$  be an undirected and edge-weighted graph. A  $t$ -spanner in  $G$  is a subgraph  $G' = (V, E')$  of  $G$  such that the shortest path between any pair of nodes  $u, v \in V$  is at most  $t$  times longer in  $G'$  than in  $G$  (where  $t > 1$ ) [22]. The NP-hard minimum-weight  $t$ -spanner problem (MWSP) is to construct a  $t$ -spanner with minimum total weight [3].

Spanners — and algorithms for their construction — form important building blocks in the design of efficient algorithms for geometric problems [10, 14, 19, 23]. Also, low-weight spanners have recently found interesting practical applications in areas such as metric space searching [21] and broadcasting in communication networks [11]. A spanner can be used as a compact data structure for holding information about (approximate) distances between pairs of objects in a large metric space, say, a collection of electronic documents; by using a spanner instead of a full distance matrix, significant space reductions can be obtained when using search algorithms like AESA [21]. For message distribution in networks, spanners can simultaneously offer both low cost and low delay when compared to existing alternatives such as minimum spanning trees (MSTs) and shortest path trees. Experiments with constructing spanners for realistic communication networks show that spanners can achieve a cost that is close to the cost of a MST while significantly reducing delay (or shortest paths between pairs of nodes) [11].

The classical algorithm for constructing a low-weight  $t$ -spanner for a graph  $G = (V, E)$  is the *greedy* algorithm [1]. All practical applications of spanners use

some variant of this algorithm. The greedy algorithm first constructs a graph  $G' = (V, E')$  where  $E' = \emptyset$ . Then the set of edges  $E$  is sorted by non-decreasing weight and the edges are processed one by one. An edge  $e = (u, v)$  is appended to  $E'$  if the weight of a shortest path in  $G'$  between  $u$  and  $v$  exceeds  $t \cdot c_e$ , where  $c_e$  is the weight of edge  $e$ . (Note that the greedy algorithm is clearly a polynomial-time algorithm; for large values of  $t$  the algorithm is identical to Kruskal's algorithm for constructing MSTs.)

The output of the greedy algorithm is a so-called *greedy spanner*. A greedy spanner has several nice theoretical properties, which are summarized in Section 2. However, it is not known how well the greedy algorithm actually performs in practice, i.e., how well the greedy spanner approximates the minimum-weight spanner for realistic problem instances.

In this paper we present the first *exact* algorithm for MWSP. We give an integer programming formulation based on column generation (Section 3). The problem decomposes into a master problem which becomes a set covering like problem and a number of subproblems which are constrained shortest path problems. The integer program is solved by branch-and-bound with lower bounds obtained by linear programming relaxations.

We have performed experiments with the new exact algorithm on a range of problem instances (Section 4). Our focus has been on comparing greedy spanners with minimum-weight spanners (or lower bounds on their weights). The results show that the greedy spanner is a surprisingly good approximation to a minimum-weight spanner — on average within 5% from optimum for the set of problem instances considered. Our conclusions and suggestions for further work are given in Section 5.

## 2 Greedy Spanner Algorithm

Let  $n = |V|$  and  $m = |E|$  for our given graph  $G = (V, E)$ . For general graphs it is hard to approximate MWSP within a factor of  $\Omega(\log n)$  [9]. However, for special types of graphs much better approximations can be obtained. Most of these are obtained by (variants of) the greedy algorithm.

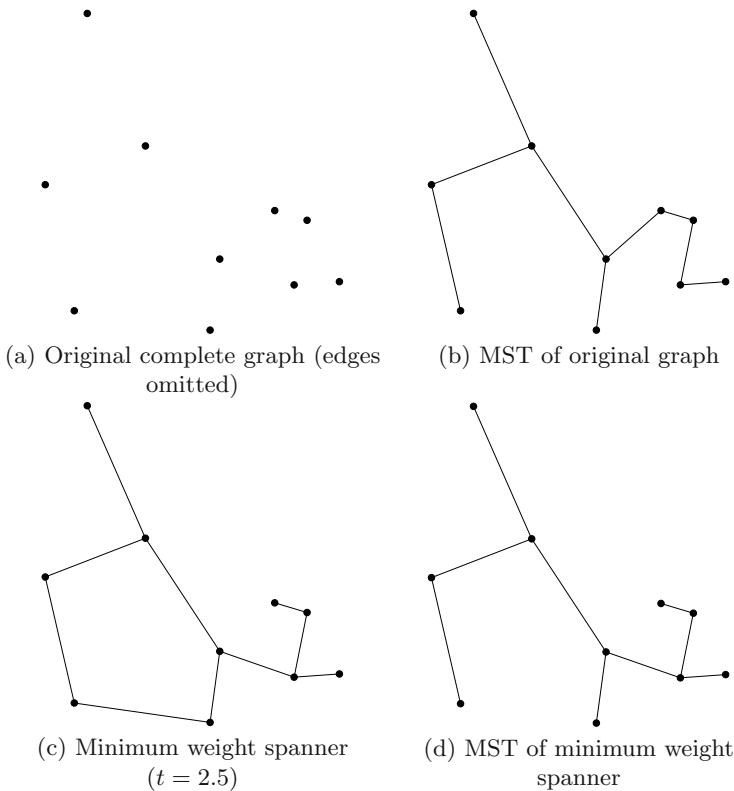
The performance of the greedy algorithm depends on the weight function on the edges of  $G$ . For general graphs, Althöfer et al. [1] showed that the greedy algorithm produces a graph with  $O(n^{1+2/(t-1)})$  edges; Chandra et al. [5] showed that the weight is bounded by  $n^{(2+\epsilon)/(t-1)}$  times that of a minimum spanning tree (MST) for  $G$  (for any  $\epsilon > 0$ ). These bounds improve to  $n(1+O(1/t))$  and  $1+O(1/t)$ , respectively, when  $G$  is planar. When the nodes of  $G$  correspond to points in some fixed dimensional space (and edge weights are equal to the Euclidean distance between the nodes), the number of edges in the greedy spanner is  $O(n)$  and the weight  $O(1)$  times that of a MST; the constant implicit in the  $O$ -notation depends on  $t$  and the dimension of the space [6, 7].

Although the bounds for the greedy algorithm are promising for planar and geometric graphs, to the best of our knowledge, little is known about the tight-

ness of these bounds and on the practical performance of the greedy algorithm on realistic problem instances.

A naive implementation of the greedy algorithm runs in  $O(n^3 \log n)$  time. Significant improvements have been made, both from the theoretical and practical side, on this running time. Gudmundsson et al. [13] gave a  $O(n \log n)$  variant of the greedy algorithm for points in fixed dimension. Navarro and Paredes [20] gave several practical variants of the greedy algorithm for general graphs with running times down to  $O(nm \log m)$ . They were able to compute approximative greedy spanners for graphs with several thousand nodes in a few minutes.

Finally, we note that a greedy spanner will always contain a MST for  $G$  [22]. On the contrary, a minimum-weight spanner need not contain a MST for the graph. An example is shown in Figure 1.



**Fig. 1.** An example showing that a minimum-weight spanner need not contain a MST of the original graph.



### 3 Column Generation Approach

In the new exact algorithm we consider the following generalization of MWSP. We are given a connected undirected graph  $G = (V, E)$  with edge weights  $c_e$ ,  $e \in E$ , a set of node pairs  $K = \{(u_i, v_i)\}$ ,  $i = 1, \dots, k$ , and a stretch factor  $t > 1$ . Let  $p_{u_i v_i}$  denote a shortest path in  $G$  between  $u_i$  and  $v_i$ ,  $(u_i, v_i) \in K$ , and let  $c(p_{u_i v_i})$  denote the length of  $p_{u_i v_i}$ . The (generalized) minimum-weight spanner problem (MWSP) consists of finding a minimum cost subset of edges  $E' \subseteq E$  such that the shortest paths  $p'_{u_i v_i}$ ,  $i = 1, \dots, k$ , in the new graph  $G' = (V, E')$  are no longer than  $t \cdot c(p_{u_i v_i})$ . Note that this problem reduces to the normal MWSP when  $K = V \times V$ .

Obviously, if  $t$  is sufficiently large, an optimal solution to the MWSP will be a minimal spanning forest. On the other hand, if  $t$  is close to 1, an optimal solution will include all edges on the shortest paths  $p_{u_i v_i}$  of  $G$ .

Our exact algorithm for solving MWSP is based on modeling the problem as an integer program, and solving this integer program by branch and bound using bounds from linear programming relaxations. MWSP can be modeled in several different ways, but the shortest path constraints are particularly difficult to model efficiently. Therefore, we have chosen a model in which *paths* are decision variables.

Given an instance of the MWSP, let  $P_{uv}$  denote the *set* of paths between  $u$  and  $v$  with length smaller than or equal to  $t \cdot c(p_{uv})$ ,  $\forall (u, v) \in K$ , and let  $P = \bigcup_{(u,v) \in K} P_{uv}$ . Furthermore, let the indicator variables  $\delta_p^e$  be defined as follows:  $\delta_p^e = 1$ , if edge  $e \in E$  is on path  $p \in P$  and  $\delta_p^e = 0$  otherwise. Let the decision variables  $x_e$ ,  $e \in E$ , equal 1 if edge  $e$  is part of the solution and 0 otherwise, and let decision variables  $y_p$ ,  $p \in P_{uv}$ , equal 1 if path  $p$  connects  $u$  and  $v$  in the solution and 0 otherwise,  $\forall (u, v) \in K$ . Then MWSP can be formulated as follows:

$$\text{minimize } \sum_{e \in E} c_e x_e \quad (1.1)$$

$$\text{subject to } \sum_{p \in P_{uv}} y_p \delta_p^e \leq x_e \quad \forall e \in E, \forall (u, v) \in K \quad (1.2)$$

$$\sum_{p \in P_{uv}} y_p \geq 1 \quad \forall (u, v) \in K \quad (1.3)$$

$$x_e \in \{0, 1\} \quad \forall e \in E \quad (1.4)$$

$$y_p \in \{0, 1\} \quad \forall p \in P \quad (1.5)$$

The objective function (1.1) minimizes the total cost of the edges in the spanner. Constraints (1.2) require that for given a pair of nodes  $(u, v)$ , all edges on the selected path connecting  $u$  and  $v$  must be part of the spanner. Constraints (1.3) say that every pair of nodes must be connected by at least one path.

We will solve the model with a branch and bound algorithm using the linear relaxation lower bound, which is obtained from (1.1)–(1.5) by relaxing the constraints (1.4) and (1.5). This model contains  $|E| + |P|$  variables and  $(|E| + 1)|K|$

constraints. Since the number of variables may be exponential in the input size, we will not solve the relaxation of (1.1)–(1.5) directly, in fact we will not even write up the model. Instead, we will solve the *restricted master problem* (RMP):

$$\begin{aligned}
 & \text{minimize} && \sum_{e \in E} c_e x_e \\
 & \text{subject to} && \sum_{p \in P'_{uv}} y_p \delta_p^e \leq x_e && \forall e \in E, \forall (u, v) \in K \\
 & && \sum_{p \in P'_{uv}} y_p \geq 1 && \forall (u, v) \in K \\
 & && x_e \geq 0 && \forall e \in E \\
 & && y_p \geq 0 && \forall p \in P'
 \end{aligned} \tag{2}$$

where  $P'_{uv} \subseteq P_{uv}$ ,  $P'_{uv} \neq \emptyset$ ,  $\forall (u, v) \in K$  and  $P' = \bigcup_{(u,v) \in K} P'_{uv}$ . To begin with, it is sufficient that  $P'_{uv}$  contains only one path for every  $(u, v) \in K$ . Iteratively, we will add paths to  $P'$ , extending the RMP. This procedure is known as *delayed column generation*. We use the Simplex method to solve the RMP in every iteration of the delayed column generation, which provides us with an optimal dual vector of the RMP. Now, consider the dual linear program of the full primal problem with dual variables  $\pi_e^{uv}$  and  $\sigma_{uv}$ ,  $\forall (u, v) \in K$ ,  $\forall e \in E$ :

$$\begin{aligned}
 & \text{maximize} && \sum_{(u,v) \in K} \sigma_{uv} \\
 & \text{subject to} && \sum_{(u,v) \in K} \pi_e^{uv} \leq c_e && \forall e \in E \\
 & && - \sum_{e \in E} \delta_p^e \pi_e^{uv} + \sigma_{uv} \leq 0 && \forall p \in P_{uv}, \forall (u, v) \in K \\
 & && \pi_e^{uv} \geq 0 && \forall e \in E, \forall (u, v) \in K \\
 & && \sigma_{uv} \geq 0 && \forall (u, v) \in K
 \end{aligned} \tag{3}$$

If the dual values obtained by solving the RMP constitute a feasible solution to (3), the weak duality theorem tells us that dual solution is an optimal solution to (3), which implies that the optimal solution to the RMP is also an optimal solution to the full problem. Otherwise the dual solution to the RMP violates one of the constraints  $-\sum_{e \in E} \delta_p^e \pi_e^{uv} + \sigma_{uv} \leq 0$  of (3) for some  $p \in P$ . The amount of violation is called the *reduced cost* of  $p \in P$  with respect to  $\pi$  and  $\sigma$ , denoted  $c_p^{\pi, \sigma}$ :

$$c_p^{\pi, \sigma} = \sum_{e \in E} \delta_p^e \pi_e^{uv} - \sigma_{uv}, \quad \forall p \in P_{uv}, \forall (u, v) \in K. \tag{4}$$

If  $c_p^{\pi, \sigma} \geq 0$ ,  $\forall p \in P$  the dual vector constitutes a feasible solution to (3), which in turn means that the current primal solution to the RMP is an optimal solu-

tion for the full master problem. In this case we may stop the delayed column generation procedure. Otherwise we add a path  $p$  with  $c_p^{\pi, \sigma} < 0$  to the RMP, which corresponds to adding a violated dual constraint to the dual of the RMP. Hence, in every iteration of the delayed column generation procedure we solve problem  $\min_{p \in P} c_p^{\pi, \sigma} = \min_{p \in P} \sum_{e \in E} \delta_p^e \pi_e^{uv} - \sigma_{uv}$ , which is called the *pricing problem*. For a given  $(u, v) \in K$ ,  $\sigma_{uv}$  is a constant and hence the pricing problem is the problem of finding a shortest path  $p \in P_{uv}$  between  $u$  and  $v$  with edge weights  $\pi_e^{uv}$ . This problem is recognized as the *constrained shortest path problem* (CSPP), since we require that  $p \in P_{uv}$ , i.e., that the length of  $p$  is no longer than  $t$  times the length of the shortest path between  $u$  and  $v$  with respect to the original edge costs.

### 3.1 Constrained Shortest Path Problem

The (resource) constrained shortest path problem (CSPP) is a generalization of the classic shortest path problem, where we impose a number of additional resource constraints. Even in the case of a single resource constraint the problem is weakly NP-complete [12]. The problem is formally stated as follows: Given a graph  $G = (V, E)$  with a weight  $w_e \geq 0$  and a cost  $c_e \geq 0$  on every edge  $e \in E$ , a source  $s \in V$  and a target  $d \in V$  and a resource limit  $B$ , find a shortest path from  $s$  to  $d$  with respect to the *cost* of the edges, satisfying that the sum of the *weights* on the path is at most  $B$ .

Several algorithms have been proposed for its solution (see e.g. [28] for a survey). Algorithms based on dynamic programming were presented in [2, 17]. A *labelling* algorithm that exploits the dominance relation between paths (i.e. path  $p$  dominates path  $q$  if it has cost  $c_p \leq c_q$  and weight  $w_p \leq w_q$ ) was proposed in Desrochers and Soumis [8]. Lagrangean relaxation has been used successfully to move the difficult resource constraint into the objective function in a two phase method, first solving the Lagrangean relaxation problem and then closing the duality gap [15, 28]. The CSPP can be approximated to any degree by polynomial algorithms by the *fully polynomial time approximation scheme* (FPTAS) given by Warburton [26].

We use a labelling algorithm similar to Desrochers and Soumis [8] with the addition of a method of early termination of infeasible paths. Our algorithm grows paths starting from the source  $s$  until we reach the target  $d$ . In the algorithm we represent a path  $p_i$  by a label  $(p_i, n_i, w_i, c_i)$ , stating that the path  $p_i$  ending in node  $n_i$  has total weight  $w_i$  and total cost  $c_i$ . If two paths  $p_1, p_2$  end in the same node,  $p_1$  dominates  $p_2$  if  $w_1 \leq w_2$  and  $c_1 \leq c_2$ . We may discard all dominated labels since they cannot be extended to optimal paths.

In our algorithm, we store the labels (paths) in a heap  $H$  ordered by the cost of the path. Iteratively, we pick the cheapest path from the heap and extend it along all edges incident to its endpoint. If the extended path does not violate the maximum weight constraint, we create a new label for this path. We remove all dominated labels, both labels generated earlier that are dominated by the new labels and new labels which are dominated by labels generated earlier. This is done efficiently by maintaining a sorted list for every node  $n$  consisting of all

labels for node  $n$ . When creating a new label, we only need to run through this list to remove dominated labels. When we reach the target node  $d$ , we know we have created the cheapest path satisfying the maximum weight constraint since we pick the cheapest label in every iteration.

The algorithm above is the standard labelling algorithm. Before we start the labelling algorithm we do two shortest path computations, creating a shortest path tree  $T^c$  from  $d$  with respect to edge costs  $c_e$  and a shortest path tree  $T^w$  from  $d$  with respect to edge weights  $w_e$ . Whenever we extend a path to a node  $n$ , we check if the new label  $(p, n, w, c)$  satisfies  $w + T^w(n) \leq B$  and  $c + T^c(n) \leq M$ , where  $M$  is the maximum cost constraint imposed on the problem (note that we are only interested in finding paths with reduced cost  $c_p^{\pi, \sigma} \leq 0$ ). If one of the inequalities is not satisfied we can discard this path, since it will not be able to reach  $d$  within the maximum weight and cost limits. The algorithm is sketched below.

CSPP( $G, w, c, B, M, s, d$ )

1. Compute shortest path trees  $T^w$  and  $T^c$ .
2.  $H \leftarrow (\{\}, s, 0, 0)$
3. Pick a cheapest label  $(p, n, w, c)$  in  $H$ . If  $n = d$  then we are done,  $p$  is the cheapest path satisfying the maximum weight constraint.
4. For all nodes  $n_i$  adjacent to  $n$  by edge  $e_i$ , extend the path and create new labels  $(p \cup \{e_i\}, n_i, w + w_{e_i}, c + c_{e_i})$ . Discard new labels where  $w + w_{e_i} + T^w(n_i) > B$  or  $c + c_{e_i} + T^c(n_i) > M$ . Discard dominated labels. Goto 3.

In every iteration of the column generation procedure, we run the CSPP algorithm for every pair of nodes  $(u, v)$  with edge costs  $\pi_e^{uv}$  and edges weights  $c_e$ , maximum cost limit  $M = \sigma^{uv}$  and maximum weight limit  $B = t \cdot c(p_{uv})$ . Note that it is not possible to run an “all constrained shortest path” algorithm, since the edge costs  $\pi_e^{uv}$  differ for every pair of nodes.

### 3.2 Implementation Details

Delayed column generation allows us to work on a LP which only contains a small number of variables. However, our restricted master problem (2) still contains a large number of constraints (for complete graphs there will be  $O(n^4)$  constraints). The constraints are all needed to ensure primal feasibility of the model, but in practice only a small number of the constraints will be active. We will remove the constraints of the form  $\sum_{p \in P_{uv}} y_p \delta_p^e \leq x_e$  from the RMP to reduce the size of the master LP. This will allow infeasible solutions, but we will check if any violated constraints exist and add these constraints to the model.

Checking whether a constraint is violated is straightforward: for all node pairs  $(u, v) \in K$  and all edges  $e \in E$ , we compute the sum  $\sum_{p \in P_{uv}} y_p \delta_p^e$  and check whether this quantity is greater than  $x_e$ . If so, we add the constraint  $\sum_{p \in P_{uv}} y_p \delta_p^e \leq x_e$  to the model. Computational experiments show that we only need to add a small fraction of the constraints to the model. This greatly speeds up the Simplex iterations in the algorithm.

Branching is done by selecting an edge variable  $x_e$  with maximum value in the fractional solution and demanding  $x_e = 1$  on one branch and  $x_e = 0$  on the other. On the  $x_e = 0$  branch the pricing problem changes a little bit, since edge  $e$  is not allowed in any of the paths generated. This can be handled very efficiently by deleting the edge from the graph on which the pricing algorithm runs. The  $x_e = 1$  branch does not change the pricing problem. We evaluate the branch and bound nodes in depth-first order.

We have used the general branch-and-cut framework ABACUS [25] with CPLEX [16] in the implementation of our algorithm.

## 4 Experimental Results

One of the main contributions of this paper is to evaluate the quality of the greedy spanner algorithm on a set of relevant problem instances. We have created two types of graphs, denoted *Euclidean* graphs and *realistic* graphs, respectively.

The *Euclidean* graphs are generated as follows. The vertices correspond to points that are randomly and uniformly distributed in a  $k$ -dimensional hypercube. Then a complete Euclidean graph is constructed from the set of points. Graphs classes for all combinations of dimensions  $k = 5, 10, 20, 25$  and  $n = 20, 30, 40, 50$  have been generated. For every class of graphs, we have created 50 instances, which we have solved by both the greedy spanner algorithm and our exact method with stretch factor  $t = 1.2, 1.4, 1.6, 1.8$ . (Note that this type of graphs is identical to the type of graphs analyzed in [4]).

The *realistic* graphs originate from the application in broadcasting and multicasting in communications networks, where spanners may be used to reduce the cost of sending data while preserving low delay. We have created a set of instances following the design proposed in [11, 27]. For  $n = 16, 32, 64$  we place  $n$  nodes uniformly random in a square. We add edges according to one of two different schemes:

1. Place edges completely at random where the probability of adding an edge is the same for all pairs of nodes.
2. Favour local edges so that the probability of adding an edge between  $u$  and  $v$  is  $\alpha e^{-d/\beta L}$ , where  $d$  is the distance between  $u$  and  $v$  and  $L$  is the diameter of the square.

By varying the probability in method 1, and  $\alpha$  and  $\beta$  in method 2, we generate graphs with average node degree  $D = 4$  and  $D = 8$ , respectively. To model realistic networks, we keep  $\beta = 0.14$  fixed and set  $\alpha$  to get the desired average node degree for  $n = 16, 32, 64$  [11].

We assign a cost to every edge in the graph according to one of three methods:

1. Unit cost. All edges have cost 1.
2. Euclidean distance. All edges have cost equal to the Euclidean distance between the endpoints.
3. Random cost. Edge cost is assigned at random from the set  $\{1, 2, 4, 8, 16\}$ .

**Table 1.** Average excess from optimum of greedy spanner (in percent) for Euclidean graphs.

# Nodes	20				30				40				50			
Dimension	5	10	20	25	5	10	20	25	5	10	20	25	5	10	20	25
$t = 1.2$	0.90	0.00	0.00	0.00	1.63	0.00	0.00	0.00	2.50	0.01	0.00	0.00	3.10	0.01	0.00	0.00
$t = 1.4$	7.61	0.83	0.00	0.00	10.88	1.20	0.01	0.00	13.00	1.43	0.00	0.00	<i>13.59</i>	1.75	0.01	0.00
$t = 1.6$	16.22	9.55	0.62	0.31	<i>20.11</i>	13.36	1.00	0.29	-	<i>15.39</i>	1.38	0.40	-	<i>18.44</i>	1.50	0.41
$t = 1.8$	21.99	33.03	18.45	13.68	-	<i>47.93</i>	<i>32.97</i>	<i>17.97</i>	-	-	-	-	-	-	-	-

**Table 2.** Average excess from optimum of greedy spanner (in percent) for realistic graphs. Results for  $\div$  *locality* and  $+$  *locality* refer to graphs where edges are added according to edge addition scheme 1 and 2, respectively, as explained in the text.

Edge cost	Unit						Euclidean						Random					
Avg. degree	4			8			4			8			4			8		
# Nodes	16	32	64	16	32	64	16	32	64	16	32	64	16	32	64	16	32	64
$t = 1.1$																		
$\div$ locality	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.02	0.02	0.57	0.23	0.14	0.00	0.00	0.00	0.00	0.00	0.00
$+$ locality	0.00	0.00	0.00	0.00	0.00	0.00	0.41	0.12	0.09	1.43	1.40	1.00	0.00	0.00	0.00	0.00	0.00	0.00
$t = 2.0$																		
$\div$ locality	7.63	<i>5.35</i>	-	-	-	-	3.25	2.22	1.29	4.71	8.48	4.35	2.76	1.17	2.46	4.22	3.29	3.33
$+$ locality	16.84	<i>15.30</i>	-	-	-	-	3.84	<i>2.73</i>	2.32	4.85	-	-	1.74	0.90	1.67	5.74	4.34	<i>3.71</i>
$t = 4.0$																		
$\div$ locality	<i>13.45</i>	-	-	-	-	-	1.31	<i>6.29</i>	-	2.10	-	-	0.64	<i>4.72</i>	-	2.64	-	-
$+$ locality	<i>9.66</i>	-	-	-	-	-	1.43	-	-	-	-	-	0.00	-	-	3.95	-	-

For every class of graphs, we have created 50 instances, which we have solved by both the greedy spanner algorithm and our exact method with stretch factor  $t = 1.1, 2.0, 4.0$ . The tests have been carried out on a 933 MHz Intel Pentium III computer, allowing each instance 30 minutes of CPU time.

In Tables 1 and 2 we show the greedy spanner’s average excess from optimum (in percent). Note the slightly different layout of these two tables, in particular with respect to the number of nodes. For some instances we have compared the greedy spanner with a lower bound on the minimum weight spanner, since we did not obtain the optimal value within the CPU time limit. This means that the true average excess from optimum may be a bit lower than stated in the table; the results for these test classes are written in italics. Some of the problem classes have not been solved since the exact method was too time consuming.

The greedy algorithm performs significantly worse on Euclidean graphs than on realistic graphs. Also, the performance of the greedy algorithm decreases as the stretch factor increases. Interestingly, the greedy spanner does not necessarily become worse for larger problem instances; for almost all test classes considered, the average excess from optimum decreases as the problem instance increases. When local edges are favored in the realistic problem instances, the quality of the greedy spanner decreases slightly for unit and Euclidean costs.

The running times depend on the size of the problem, the edge cost function and the stretch factor. Naturally, the spanner problem becomes harder to solve for larger problem instances. It turns out that spanner problems with unit edge costs are considerably harder to solve by our exact method than spanner problems with Euclidean or random costs. The difficulties for problem instances with unit cost are caused by the large number of paths of equal cost, which makes it hard for the branch and bound algorithm to choose the best path. Thus, we need to evaluate more branch and bound nodes to solve unit cost problem instances.

We experienced that our solution times increased when the stretch factor increased. This is because the set of feasible paths between any two nodes increases, potentially increasing the number of columns in our master problem. On 75% of the test instances the lower bound provided in the root node was optimal. On the remaining 25% of the test instances, the lower bound in the root node was 6.4% from the optimum on average. This shows the quality of the proposed integer programming formulation.

## 5 Conclusions

In this paper we presented a first exact algorithm for the MWSP. Experiments on a set of realistic problem instances from applications in message distribution in networks were reported. Problem instances with up to 64 nodes have been solved to optimality. The results show that the total weight of a spanner constructed by the greedy spanner algorithm is typically within a few percent from optimum for the set of realistic problem instances considered. Thus the greedy spanner algorithm appears to be an excellent choice for constructing approximate minimum-weight spanners in practice.

### 5.1 Generalizations of the MWSP

The minimum-weight spanner problem may be seen as a special case of a larger class of network problems with shortest path constraints. This class of problems is normally hard to model due to the difficult shortest path constraints.

The exact method we present in this paper can easily be modified to solve this larger class of network problems with shortest path constraints. Here we describe how our method can be applied to several generalizations of the MWSP.

**MWSP with variable stretch factor.** Consider the MWSP as defined above, but with variable stretch factor for every pair of nodes. This is a generalization of the MWSP. Our method can easily be modified to solve this problem, since we already generate different variables for every pair of nodes satisfying the maximum cost constraint. Thus, it is possible to allow different cost constraints for every pair of nodes.

**MWSP with selected shortest path constraints.** Consider the MWSP where we do not have maximum length constraints on all pairs of nodes. The solution to this problem may be a disconnected graph consisting of several smaller spanners. This generalization can also be used to model problems

where there are maximum cost constraints between a root node and all other nodes [18]. Our method can be modified to solve this problem by removing the covering constraints (1.3) for pairs of nodes  $(u, v)$  which have no shortest path constraints imposed.

## 5.2 Future Work

The main focus in this paper has been to evaluate the quality of the greedy algorithm, and to present a framework for a promising exact algorithm. It would be interesting to evaluate our algorithm on a wider set of problem instances, and to examine the practical performance of other heuristic spanner algorithms, including variants of the greedy algorithm (e.g., in which simple local improvements are made in the greedy spanner),  $\Theta$ -graphs and well-separated pair decompositions.

Clearly, there is still room for improvements on the exact algorithm. Providing our exact method with a good upper bound computed by, e.g., the greedy algorithm will probably help to prune the branch and bound tree. By adding more variables in each column generation iteration the procedure may be accelerated. Finally, column generation stabilization [24] will most likely have an accelerating effect.

## References

- [1] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On Sparse Spanners of Weighted Graphs. *Discrete and Computational Geometry*, 9:81–100, 1993.
- [2] Y. P. Aneja, V. Aggarwal, and K. P. K. Nair. Shortest Chain Subject to Side Constraints. *Networks*, 13:295–302, 1983.
- [3] L. Cai. NP-Completeness of Minimum Spanner Problem. *Discrete Applied Mathematics*, 48:187–194, 1994.
- [4] B. Chandra. Constructing Sparse Spanners for Most Graphs in Higher Dimension. *Information Processing Letters*, 51:289–294, 1994.
- [5] B. Chandra, G. Das, G. Narasimhan, and J. Soares. New Sparseness Results on Graph Spanners. *International Journal of Computational Geometry and Applications*, 5:125–144, 1995.
- [6] G. Das and G. Narasimhan. A Fast Algorithm for Constructing Sparse Euclidean Spanners. *Int. J. Comput. Geometry Appl.*, 7(4):297–315, 1997.
- [7] G. Das, G. Narasimhan, and J.S. Salowe. A New Way to Weigh Malnourished Euclidean Graphs. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 215–222, 1995.
- [8] M. Desrochers and F. Soumis. A Generalized Permanent Labelling Algorithm for the Shortest Path Problem with Time Windows. *INFOR*, 26:191–211, 1988.
- [9] Y. Dodis and S. Khanna. Design Networks with Bounded Pairwise Distance. In *Proceedings of the 31th Annual ACM Symposium on Theory of Computing*, pages 750–759, 1999.
- [10] D. Eppstein. Spanning Trees and Spanners. In *Handbook of Computational Geometry*, pages 425–461, 1999.
- [11] A. M. Farley, D. Zappala A. Proskurowski, and K. Windisch. Spanners and Message Distribution in Networks. *Discrete Applied Mathematics*, 137:159–171, 2004.



- [12] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W.H. Freeman and Co., San Francisco, 1979.
- [13] J. Gudmundsson, C. Levcopoulos, and G. Narasimhan. Fast Greedy Algorithms for Constructing Sparse Geometric Spanners. *SIAM Journal on Computing*, 31(5):1479–1500, 2002.
- [14] J. Gudmundsson, C. Levcopoulos, G. Narasimhan, and M. Smid. Approximate Distance Oracles for Geometric Graphs. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 828–837, 2002.
- [15] G. Y. Handler and I. Zang. A Dual Algorithm for the Constrained Shortest Path Problem. *Networks*, 10:293–310, 1980.
- [16] ILOG. *ILOG CPLEX 7.0, Reference Manual*. ILOG, S.A., France, 2000.
- [17] H.C. Joksche. The Shortest Route Problem with Constraints. *J. Math. Anal. Appl.*, 14:191–197, 1966.
- [18] S. Khuller, B. Raghavachari, and N. Young. Balancing Minimum Spanning and Shortest-Path Trees. *Algorithmica*, 14(4):305–321, 1995.
- [19] G. Narasimhan and M. Zachariasen. Geometric Minimum Spanning Trees via Well-Separated Pair Decompositions. *ACM Journal of Experimental Algorithmics*, 6, 2001.
- [20] G. Navarro and R. Paredes. Practical Construction of Metric  $t$ -Spanners. In *Proceedings of the 5th Workshop on Algorithm Engineering and Experiments (ALENEX'03)*, 2003.
- [21] G. Navarro, R. Paredes, and E. Chavez.  $t$ -Spanners as a Data Structure for Metric Space Searching. In *International Symposium on String Processing and Information Retrieval, SPIRE, LNCS 2476*, pages 298–309, 2002.
- [22] D. Peleg and A. Schaffer. Graph Spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.
- [23] S. B. Rao and W. D. Smith. Improved Approximation Schemes for Geometrical Graphs via "Spanners" and "Banyans". In *Proceedings 30th Annual ACM Symposium on Theory of Computing*, pages 540–550, 1998.
- [24] M. Sigurd and D. Ryan. Stabilized Column Generation for Set Partitioning Optimization. In preparation, 2003.
- [25] S. Thienel. *ABACUS — A Branch-And-Cut System*. PhD thesis, Universität zu Köln, Germany, 1995.
- [26] A. Warburton. Approximation of Pareto Optima in Multiple-Objective, Shortest Path Problems. *Operations Research*, 35:70–79, 1987.
- [27] B. M. Waxman. Routing of Multipoint Connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–22, 1988.
- [28] M. Ziegelmann. *Constrained Shortest Paths and Related Problems*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2001.

# A Straight Skeleton Approximating the Medial Axis

Mirela Tănase and Remco C. Veltkamp

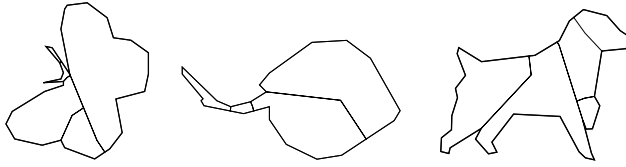
Institute of Information & Computing Sciences  
Utrecht University, The Netherlands  
{mirela, Remco.Veltkamp}@cs.uu.nl

**Abstract.** We propose the linear axis, a new skeleton for polygonal shapes. It is related to the medial axis and the straight skeleton, being the result of a wavefront propagation process. The wavefront is linear and propagates by translating edges at constant speed. The initial wavefront is an altered version of the original polygon: zero-length edges are added at reflex vertices. The linear axis is a subset of the straight skeleton of the altered polygon. In this way, the counter-intuitive effects in the straight skeleton caused by sharp reflex vertices are alleviated. We introduce the notion of  $\varepsilon$ -equivalence between two skeletons, and give an algorithm that computes the number of zero-length edges for each reflex vertex which yield a linear axis  $\varepsilon$ -equivalent to the medial axis. This linear axis and thus the straight skeleton can be computed from the medial axis in linear time for polygons with a constant number of “nearly co-circular” sites. All previous algorithms for straight skeleton computation are sub-quadratic.

## 1 Introduction

Skeletons have long been recognized as an important tool in computer graphics, computer vision and medical imaging. The most known and widely used skeleton is the *medial axis* [1]. Variations of it include *smoothed local symmetries* [2] and the *process inferring symmetric axis* [3]. One way to define the medial axis of a polygon  $P$  is as the locus of centers of maximally inscribed disks. Another way to describe it is as the locus of singularities of a propagating wavefront, whose points move at constant, equal speed. The medial axis is also a subset of the Voronoi diagram whose sites are the line segments and the reflex vertices of  $P$ . The medial axis has always been regarded as an attractive shape descriptor. However, the presence of parabolic arcs may constitute a disadvantage for the medial axis. Various linear approximations were therefore proposed. One approach is to use the Voronoi diagram of a sampled set of boundary points [4].

The straight skeleton is defined only for polygonal figures and was introduced by Aichholzer and Aurenhammer [5]. As its name suggests, it is composed only of line segments. Also the straight skeleton has a lower combinatorial complexity than the medial axis. Like the medial axis, the straight skeleton is defined by a



**Fig. 1.** A decomposition [7] based on split events of the straight line skeleton gives counter-intuitive results if the polygon contains sharp reflex vertices.

wavefront propagation process. In this process, edges of the polygon move inward at a fixed rate.

In [7] a polygon decomposition based on the straight skeleton was presented. The results obtained from the implementation indicated that this technique provides plausible decompositions for a variety of shapes. However, sharp reflex angles have a big impact on the form of the straight skeleton, which in turn has a large effect on the decomposition. Figure 1 shows three examples of the decomposition based on wavefront split events. It clearly shows that the sharp reflex vertices cause a counterintuitive decomposition. The reason why sharp reflex vertices have a big impact on the form of the straight skeleton, is that points in the defining wavefront close to such reflex vertices move much faster than other wavefront points. In contrast, points in the wavefront defining the medial axis move at equal, constant speed (but this leads to the presence of parabolic arcs).

### 1.1 Contribution

(i) This paper presents the *linear axis*, a new skeleton for polygons. It is based on a linear propagation, like in the straight skeleton. The speed of the wavefront points originating from a reflex vertex is decreased by inserting in the initial wavefront at each reflex vertex a number of zero-length edges, called *hidden edges*. In the propagation, wavefront edges translate at constant speed in a self-parallel manner. The linear axis is the trace of the convex vertices of the propagating wavefront. It is therefore a subset of the straight skeleton of an altered version of the polygon.

(ii) Properties of the linear axis are given in section 3. We compute upper bounds on the speed of the points in the propagating wavefront, that also allows us to identify localization constraints for the skeletal edges, in terms of parabola, hyperbola and Apollonius circles. These properties also prove that the larger the number of hidden edges the better the linear axis approximates the medial axis.

(iii) A thorough analysis of the relation between the number of the inserted hidden edges and the quality of this approximation is given in section 4. We introduce the notion of  $\varepsilon$ -equivalence between the two skeletons. Nodes in the two skeletons are clustered based on a proximity criterion, and the  $\varepsilon$ -equivalence between the two skeletons is defined as an isomorphism between the resulting graphs with clusters as vertices. This allows us to compare skeletons based on their main topological structure, ignoring local detail. We next give an algorithm

for computing a number of hidden edges for each reflex vertex such that the resulting linear axis is  $\varepsilon$ -equivalent to the medial axis.

(iv) This linear axis can then be computed from the medial axis. The whole linear axis computation takes linear time for polygons with a constant number of nodes in any cluster. There is only a limited category of polygons not having this property.

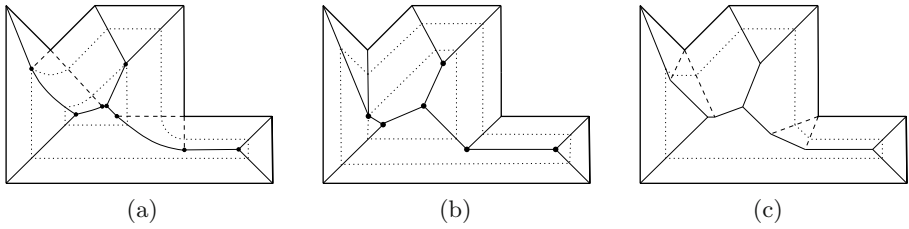
(v) Experimental verification shows that in practice only a few hidden edges are necessary to yield a linear axis that is  $\varepsilon$ -equivalent to the medial axis. Also the resulting decomposition is much more intuitive than the decomposition based on the straight skeleton.

## 2 Skeletons from Propagation

Let  $P$  be a simple, closed polygon. The medial axis  $M(P)$  is closely related to the Voronoi diagram  $VD(P)$  of the polygon  $P$ . The set of sites defining  $VD(P)$  is the set of line segments and reflex vertices of  $P$ . The diagram  $VD(P)$  partitions the interior of  $P$  into Voronoi cells, which are regions with one closest site. The set of points  $\mathcal{U}(d)$  inside  $P$ , having some fixed distance  $d$  to the polygon is called a *uniform wavefront*. If  $S$  is an arbitrary site of  $P$ , let  $\mathcal{U}_S(d)$  denote the set of points in  $\mathcal{U}(d)$  closest to  $S$ . We call  $\mathcal{U}_S(d)$  the *(uniform) offset* of  $S$ . The uniform wavefront is composed of straight line segments (offsets of line segments) and circular arcs (offsets of reflex vertices) (see figure 2(a)). As the distance  $d$  increases, the wavefront points move at equal, constant velocity along the normal direction. This process is called *uniform wavefront propagation*. During the propagation, the breakpoints between adjacent offsets trace the Voronoi diagram  $VD(P)$ . The medial axis  $M(P)$  is a subset of  $VD(P)$ ; the Voronoi edges incident to the reflex vertices are not part of the medial axis.

The region  $VC(S)$  swept in the propagation by the offset of site  $S$  is the Voronoi cell of  $S$ . It is also the set of points inside  $P$  whose closest site is  $S$ ; the distance  $d(x, S)$  of a point  $x$  to  $S$  is the length of the shortest path inside  $P$  from  $x$  to  $S$ . There are two types of edges in  $M(P)$ : line segments (separating the Voronoi cells of two line segments or two reflex vertices) and parabolic arcs (separating the Voronoi cells of a line segment and a reflex vertex). A *branching node*  $b \in M(P)$  is a node of degree at least three, and the sites whose Voronoi cells intersect in  $b$  are referred to as the *generator sites* of  $b$ . The medial axis contains also nodes of degree two, which will be referred to as the *regular nodes*. They are breakpoints between parabolic arcs and line segments.

The straight skeleton is also defined as the trace of adjacent components of a propagating wavefront. The wavefront is linear and is obtained by translating the edges of the polygon in a self-parallel manner, keeping sharp corners at reflex vertices. In contrast to the medial axis, edges incident to a reflex vertex will grow in length (see figure 2(b)). This also means that points in the wavefront move at different speeds: wavefront points originating from reflex vertices move faster than points originating from the edges incident to those vertices. In fact, the speed of the wavefront points originating from a reflex vertex gets arbitrary



**Fig. 2.** The medial axis (a), the straight skeleton (b) and the linear axis in the case when one hidden edge is inserted at each reflex vertex (c). Instances of the propagating wavefront generating the skeletons are drawn with dotted line style in all cases. In (a) the Voronoi edges that are not part of the medial axis are in dashed line style. In (b) the dashed lines are the bisectors that are not part of the linear axis.

large when the internal angle of the vertex gets arbitrary close to  $2\pi$ . Wavefront vertices move on angular bisectors of wavefront edges. The straight skeleton  $SS(P)$  of  $P$  is the trace in the propagation of the vertices of the wavefront.

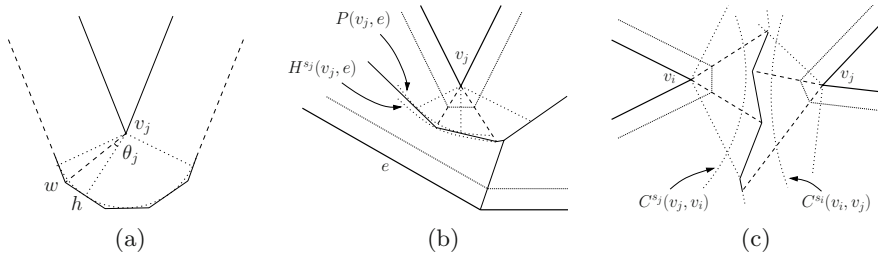
### 3 Linear Axis – Definition and Properties

In this section we define the *linear axis*. It is based on a linear wavefront propagation like the straight skeleton, but the discrepancy in the speed of the points in the propagating wavefront, though never zero, can decrease as much as wanted. Also, as it turns out from lemma 2, the speed of the points of this wavefront is bounded from above by  $\sqrt{2}$ .

Let  $\{v_1, v_2, \dots, v_n\}$  denote the vertices of a simple polygon  $P$ , and let  $\kappa = (k_1, k_2, \dots, k_n)$  be a sequence of natural numbers. If  $v_i$  is a convex vertex of  $P$ ,  $k_i = 0$ , and if it is a reflex vertex,  $k_i \geq 0$ . Let  $\mathcal{P}^\kappa(0)$  be the polygon obtained from  $P$  by replacing each reflex vertex  $v_i$  with  $k_i + 1$  identical vertices, the endpoints of  $k_i$  zero-length edges, which will be referred to as the *hidden edges* associated with  $v_i$ . The directions of the hidden edges are chosen such that the reflex vertex  $v_i$  of  $P$  is replaced in  $\mathcal{P}^\kappa(0)$  by  $k_i + 1$  “reflex vertices” of equal internal angle. Let  $\mathcal{P}^\kappa(t)$  denote the linear wavefront, corresponding to a sequence  $\kappa$  of hidden edges, at moment  $t$ . The propagation process consists of translating edges at constant unit speed, in a self-parallel manner, i.e. it is the propagation of the wavefront defining the straight skeleton of  $\mathcal{P}^\kappa(0)$ .

**Definition 1.** The *linear axis*  $L^\kappa(P)$  of  $P$ , corresponding to a sequence  $\kappa$  of hidden edges, is the trace of the convex vertices of the linear wavefront  $\mathcal{P}^\kappa$  in the above propagation process.

Obviously,  $L^\kappa(P)$  is a subset of  $SS(\mathcal{P}^\kappa(0))$ ; we only have to remove the bisectors traced by the reflex vertices of the wavefront (see figure 2 (c)). For the rest of this paper, we will assume that each selection  $\kappa$  of hidden edges that is considered, satisfies the condition in the following lemma.



**Fig. 3.** (a) The linear offset (solid line style) of a reflex vertex with 3 associated hidden edges is made of 5 segments tangent to the uniform offset (dotted line style) of this vertex. (b) The linear offsets of reflex vertex  $v_j$  and edge  $e$  trace, in the propagation, a polygonal chain that lies in the region between parabola  $P(v_j, e)$  and hyperbola  $H^{s_j}(v_j, e)$ . (c) The linear offsets of the reflex vertices  $v_i$  and  $v_j$  trace a polygonal chain that lies outside the Apollonius circles  $C^{s_j}(v_j, v_i)$  and  $C^{s_i}(v_i, v_j)$

**Lemma 1.** *If any reflex vertex  $v_j$  of internal angle  $\alpha_j \geq 3\pi/2$  has at least one associated hidden edge, then  $L^\kappa(P)$  is an acyclic connected graph.*

A site of  $P$  is a line segment or a reflex vertex of the polygon. If  $S$  is an arbitrary site of  $P$ , we denote by  $\mathcal{P}_S^\kappa(t)$  the points in  $\mathcal{P}^\kappa(t)$  originating from  $S$ . We call  $\mathcal{P}_S^\kappa(t)$  the (linear) offset of site  $S$  at moment  $t$ . Figure 3(a) illustrates the linear and the uniform offsets of a reflex vertex  $v_j$ , with  $k_j = 3$  associated hidden edges.

Individual points in  $\mathcal{P}^\kappa$  move at different speeds. The fastest moving points in  $\mathcal{P}^\kappa$  are its reflex vertices. The slowest moving points have unit speed (it also means that we assume unit speed for the points in the uniform wavefront). The next lemma gives bounds on the speed of the points in the linear wavefront  $\mathcal{P}^\kappa$ . Let  $v_j$  be a reflex vertex of  $P$ , of internal angle  $\alpha_j$ , and with  $k_j$  associated hidden edges. Let  $\mathcal{P}_{v_j}^\kappa(t)$  be the offset of  $v_j$  at some moment  $t$ .

**Lemma 2.** *Points in  $\mathcal{P}_{v_j}^\kappa(t)$  move at speed at most*

$$s_j = \frac{1}{\cos\left(\frac{\alpha_j - \pi}{2*(k_j + 1)}\right)}.$$

The linear axis  $L^\kappa(P)$  is the trace of the convex vertices of the propagating linear wavefront. Each convex vertex of  $\mathcal{P}^\kappa(t)$  is a breakpoint between two linear offsets. Lemmas 3 and 4 describe the trace of the intersection of two adjacent offsets in the linear wavefront propagation. They are central to the algorithms in section 4. First, we establish the needed terminology. If  $v$  is a vertex and  $e$  is an edge non-incident with  $v$ , we denote by  $P(v, e)$  the parabola with focus  $v$  and directrix the line supporting  $e$ . For any real value  $r > 1$ , we denote by  $H^r(v, e)$  the locus of points whose distances to  $v$  and the line supporting  $e$  are in constant ratio  $r$ . This locus is a hyperbola branch. If  $u$  and  $v$  are reflex vertices, we denote by  $C^r(u, v)$  the Apollonius circle associated with  $u$ , and  $v$ , and ratio

$r \neq 1$ .  $C^r(u, v)$  is the locus of points whose distances to  $u$  and  $v$ , respectively, are in constant ratio  $r$ .

Let  $e$  be an edge and  $v_j$  be a reflex vertex of  $P$  (see figure 3 (b)). The points in the offset  $\mathcal{P}_e^\kappa$  move at unit speed, the points in the offset  $\mathcal{P}_{v_j}^\kappa$  move at speeds varying in the interval  $[1, s_j]$ , where  $s_j$  is given by lemma 2.

**Lemma 3.** *If the linear offsets of  $v_j$  and  $e$  become adjacent, the trace of their intersection is a polygonal chain that satisfies:*

- 1) *it lies in the region between  $P(v_j, e)$  and  $H^{s_j}(v_j, e)$ ;*
- 2) *the lines supporting its segments are tangent to  $P(v_j, e)$ ; the tangent points lie on the traces of the unit speed points in  $\mathcal{P}_{v_j}^\kappa$ ;*
- 3)  *$H^{s_j}(v_j, e)$  passes through those vertices of the chain which lie on the trace of a reflex vertex of  $\mathcal{P}_{v_j}^\kappa$ .*

Let  $v_i$  and  $v_j$  be reflex vertices of  $P$  (see figure 3 (c)). The points in the offset  $\mathcal{P}_{v_i}^\kappa$  move at speeds varying in the interval  $[1, s_i]$ , while the points in the offset  $\mathcal{P}_{v_j}^\kappa$  move at speeds varying in the interval  $[1, s_j]$ .

**Lemma 4.** *If the linear offsets of  $v_i$  and  $v_j$  become adjacent, the trace of their intersection is a polygonal chain that lies outside the Apollonius circles  $C^{s_i}(v_i, v_j)$  and  $C^{s_j}(v_j, v_i)$ .*

## 4 Linear Axis, Topologically Similar to the Medial Axis

Obviously, the larger the number of hidden edges associated with the reflex vertices, the closer the corresponding linear axis approximates the medial axis. For many applications of the medial axis, an approximation that preserves the main visual cues of the shape, though perhaps non-isomorphic to the medial axis, is perfectly adequate. The way we now define the  $\varepsilon$ -equivalence between the medial axis and the linear axis, will allow us to compute a linear axis that closely approximates the medial axis using only a small number of hidden edges.

Let  $\varepsilon \geq 0$ ; an  $\varepsilon$ -edge is a Voronoi edge generated by four almost co-circular sites (see figure 4 (a)). Let  $b_i b_j$  be a Voronoi edge, with  $b_i$  generated by sites  $S_k, S_i, S_l$ , and  $b_j$  generated by  $S_k, S_j, S_l$ .

**Definition 2.** *The edge  $b_i b_j$  is an  $\varepsilon$ -edge if  $d(b_i, S_j) < (1 + \varepsilon)d(b_i, S_i)$  or  $d(b_j, S_i) < (1 + \varepsilon)d(b_j, S_j)$ .*

A path between two nodes of  $M(P)$  is an  $\varepsilon$ -path if it is made only of  $\varepsilon$ -edges. For any node  $b$  of  $M(P)$ , a node  $b'$  such that the path between  $b$  and  $b'$  is an  $\varepsilon$ -path, is called an  $\varepsilon$ -neighbour of  $b$ . Let  $N_\varepsilon(b)$  be the set of  $\varepsilon$ -neighbours of  $b$ . The set  $\{b\} \cup N_\varepsilon(b)$  is called an  $\varepsilon$ -cluster.

Let  $(V_M, E_M)$  be the graph induced by  $M(P)$  on the set of vertices  $V_M$  composed of the convex vertices of  $P$  and the nodes of degree 3 of  $M(P)$ . Let  $(V_{L^\kappa}, E_{L^\kappa})$  be the graph induced by  $L^\kappa(P)$  on the set of vertices  $V_{L^\kappa}$  composed of the convex vertices of  $P$  and the nodes of degree 3 of  $L^\kappa(P)$ .

**Definition 3.**  $M(P)$  and  $L^\kappa(P)$  are  $\varepsilon$ -equivalent if there exists a surjection  $f : V_M \rightarrow V_{L^\kappa}$  so that:

- i)  $f(p) = p$ , for all convex  $p$  of  $P$ ;
- ii)  $\forall b_i, b_j \in V_M$  with  $b_j \notin N_\varepsilon(b_i)$ ,  $\exists$  an arc in  $E_M$  connecting  $b_i$  and  $b_j \Leftrightarrow \exists$  an arc in  $E_{L^\kappa}$  connecting  $f(b'_i)$  and  $f(b'_j)$  where  $b'_i \in \{b_i\} \cup N_\varepsilon(b_i)$  and  $b'_j \in \{b_j\} \cup N_\varepsilon(b_j)$ .

The following lemma gives a sufficient condition for the  $\varepsilon$ -equivalence of the two skeletons. The path between two disjoint Voronoi cells  $VC(S_i)$  and  $VC(S_j)$  is the shortest path in  $M(P)$  between a point of  $VC(S_i) \cap M(P)$  and a point of  $VC(S_j) \cap M(P)$ .

**Lemma 5.** If the only sites whose linear offsets become adjacent in the propagation, are sites whose uniform offsets trace an edge in the medial axis  $M(P)$  or sites whose path between their Voronoi cells is an  $\varepsilon$ -path, then the linear axis and the medial axis are  $\varepsilon$ -equivalent.

#### 4.1 Computing a Sequence of Zero-Length Edges

We now describe an algorithm for computing a sequence  $\kappa$  of hidden edges such that the resulting linear axis is  $\varepsilon$ -equivalent to the medial axis. As lemma 5 indicates, the linear axis and the medial axis are  $\varepsilon$ -equivalent if only the linear offsets of certain sites become adjacent in the propagation. Namely, those with adjacent Voronoi cells, and those whose path between their Voronoi cells is an  $\varepsilon$ -path. The algorithm handles pairs of sites whose linear offsets must be at any moment disjoint in order to ensure the  $\varepsilon$ -equivalence of the two skeletons. These are sites with disjoint Voronoi cells and whose path between these Voronoi cells is not an  $\varepsilon$ -path. However, we do not have to consider each such pair. The algorithm actually handles the pairs of *conflicting sites*, where two sites  $S_i$  and  $S_j$  (at least one being a reflex vertex) are said to be conflicting if the path between their Voronoi cells contains exactly one non- $\varepsilon$ -edge. When handling a pair  $S_i, S_j$  we check and, if necessary, adjust the maximal speeds  $s_i$  and  $s_j$  of the offsets of  $S_i$  and  $S_j$ , respectively, so that these offsets remain disjoint in the propagation. This is done by looking locally at the configuration of the uniform wavefront and using the localization constraints for the edges of the linear axis given by lemmas 3 and 4. More insight into this process is provided below.

**Handling conflicting sites.** Let  $S_i$  and  $S_j$  be a pair of conflicting sites. Let  $b_i \in VC(S_i)$  and  $b_j \in VC(S_j)$  denote the endpoints of the path between their Voronoi cells. The handling of this pair depends on the composition of the path between  $VC(S_i)$  and  $VC(S_j)$ , as well as on the type (reflex vertex or segment) of  $S_i$  and  $S_j$ . A detailed case analysis of all possible combinations is not possible due to space limitations. For this reason, we present here only the structure of the case analysis and detail the handling only in two of these cases. Each of these cases requires a different handling. A complete description can be found in [8].



**A.** The path between the Voronoi cells  $VC(S_i)$  and  $VC(S_j)$  is made of one non- $\varepsilon$ -edge. Let  $S_k$  and  $S_l$  denote the sites whose uniform offsets trace this edge. When handling the pair  $S_i, S_j$  we look locally at the configuration of the uniform wavefront shortly prior to the moment the Voronoi edge  $b_i b_j$  starts being traced in the propagation. We take into consideration not only the type of the sites  $S_i$ , and  $S_j$ , but also the type (regular node or branching node) of  $b_i$  and  $b_j$ .

**A.1.**  $S_i$  is reflex vertex and  $S_j$  is line segment. Regarding  $b_i$  and  $b_j$  we have the following cases:

- $b_i$  is a branching node and  $b_j$  is a branching or a regular node of  $M(P)$  (see figure 4(b)). The handling of the pair  $S_i, S_j$  in this case is given by:

**Lemma 6.** *If the speed of the points in the linear offset of  $S_i$  is at most  $\frac{d(b_j, S_i)}{d(b_j, S_k)}$ , the linear offsets of  $S_i$  and  $S_j$  are at any moment disjoint.*

- $b_i$  is a regular node occurring sooner than  $b_j$ , which is a branching node.
- $b_i$  is a regular node occurring later than  $b_j$ , which is a regular or branching node.

**A.2.**  $S_i$  and  $S_j$  are reflex vertices. Regarding  $b_i$  and  $b_j$  we distinguish the following cases:

- $b_i$  and  $b_j$  are branching points. Let  $b$  be the intersection of the perpendicular bisector  $B_{ij}$  of  $S_i$  and  $S_j$  with the edge connecting  $b_i$  and  $b_j$  (see figure 4, (c)). The handling of the pair  $S_i, S_j$  in this case is indicated by:

**Lemma 7.** *If the speed of the points in the linear offset of  $S_i$  is at most  $\frac{d(b, S_i)}{d(b, S_k)}$ , and the speed of the points in the linear offset of  $S_j$  is at most  $\frac{d(b, S_j)}{d(b, S_k)}$ , the linear offsets of  $S_i$  and  $S_j$  are at any moment disjoint.*

- $b_i$  is a regular node occurring sooner than  $b_j$ , which is a branching node.
- $b_i$  is a regular node occurring later than  $b_j$ , which is a branching node.
- $b_i$  and  $b_j$  are regular nodes.

**B.** The path between the Voronoi cells  $VC(S_i)$  and  $VC(S_j)$  is made of at least two edges, all except one of them being  $\varepsilon$ -edges. A similar case description as above is omitted due to space limitations.

**Algorithm.** The algorithm for computing a sequence  $\kappa$  of hidden edges can now be summarized as follows.

#### Algorithm ComputeHiddenEdges ( $P, \varepsilon$ )

*Input* A simple polygon  $P$  and a constant  $\varepsilon$ .

*Output* The number of hidden edges for each reflex vertex such that the resulting linear axis is  $\varepsilon$ -equivalent to the medial axis.

1. Compute the medial axis  $\mathbf{M}$  of  $P$ .

2. For each reflex vertex  $S_j$  of  $P$

if  $\alpha_j \geq 3\pi/2$  then  $s_j \leftarrow 1/\cos(\frac{\alpha_j - \pi}{4})$

else  $s_j \leftarrow 1/\cos(\frac{\alpha_j - \pi}{2})$

3. ComputeConflictingSites( $\varepsilon$ )

4. For each pair of conflicting sites  $S_i, S_j$

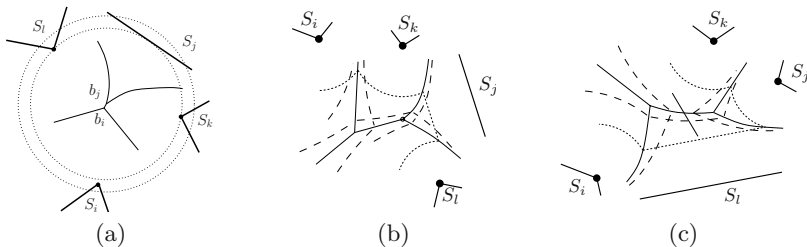
$k_j \leftarrow \lceil (\alpha_j - \pi)/(2 \cos^{-1}(1/s_j)) \rceil$ .

In step 1, for each reflex vertex  $S_j$  we initialize the maximal speed  $s_j$  of the reflex vertices in its linear offset. If the angle  $\alpha_j$  of  $S_j$  is greater than  $3\pi/2$ , we associate one hidden edge with  $v_j$ , otherwise the initial number  $k_j$  of hidden edges associated to  $S_j$  is 0 (see section 3). The value of  $s_j$  is then given by lemma 2. In handling a pair of conflicting sites  $S_i, S_j$  in step 3, we use the bounds given by lemmas similar to lemma 6 and 7. After identifying in which of the cases A.1.1-B the conflicting pair falls into, we check whether the current speeds  $s_i$  and/or  $s_j$  satisfy the condition(s) in the corresponding lemma. If they do not, we adjust  $s_i$  and/or  $s_j$  to the bound(s) given by the lemma. Finally in step 4, the number  $k_j$  of hidden edges associated with each reflex vertex  $S_j$  is determined. It is the smallest number of hidden edges such that the speed of the vertices in the linear offset of  $S_j$  is at most  $s_j$ .

Regarding the computation of the conflicting sites in step 2, we notice that each edge of  $M(P)$ , non-incident to  $P$ , is a path between two Voronoi cells. Thus each non- $\varepsilon$ -edge of  $M(P)$ , non-incident to  $P$ , defines two possibly conflicting sites. What determines whether these are indeed conflicting is the presence of at least one reflex vertex in the pair. If at least one endpoint of the non- $\varepsilon$ -edge is part of an  $\varepsilon$ -cluster, there may be other conflicting sites whose path between their Voronoi cells contains this edge. Their number depends on the number of nodes in the  $\varepsilon$ -clusters of the endpoints of the non- $\varepsilon$ -edge.

**Theorem 1.** *Algorithm ComputeHiddenEdges computes a sequence of hidden edges that leads to a linear axis  $\varepsilon$ -equivalent to the medial axis.*

The performance of the algorithm ComputeHiddenEdges depends on the number of conflicting pairs computed in step 2. This in turn depends on the number of nodes in the  $\varepsilon$ -clusters of  $M(P)$ . If any  $\varepsilon$ -cluster of  $M(P)$  has a



**Fig. 4.** (a) An  $\varepsilon$ -edge  $b_i b_j$  is a Voronoi edge generated by four almost co-circular sites. (b)-(c) Handling the conflicting sites  $S_i$  and  $S_j$  when the path between their Voronoi cells consists of one edge. An instance of the uniform wavefront is drawn in dotted line style, the medial axis in solid line style, and the dashed lines give the localization constraints for the edges in the linear axis.

constant number of nodes, there are only a linear number of conflicting pairs, since there is a linear number of non- $\varepsilon$ -edges in  $M(P)$ . Each conflicting pair is handled in constant time, thus in this case, `ComputeHiddenEdges` computes the sequence  $\kappa$  in linear time. There is only a limited class of shapes with a constant number of clusters, each with a linear number of nodes.

## 4.2 Computation of the Linear Axis

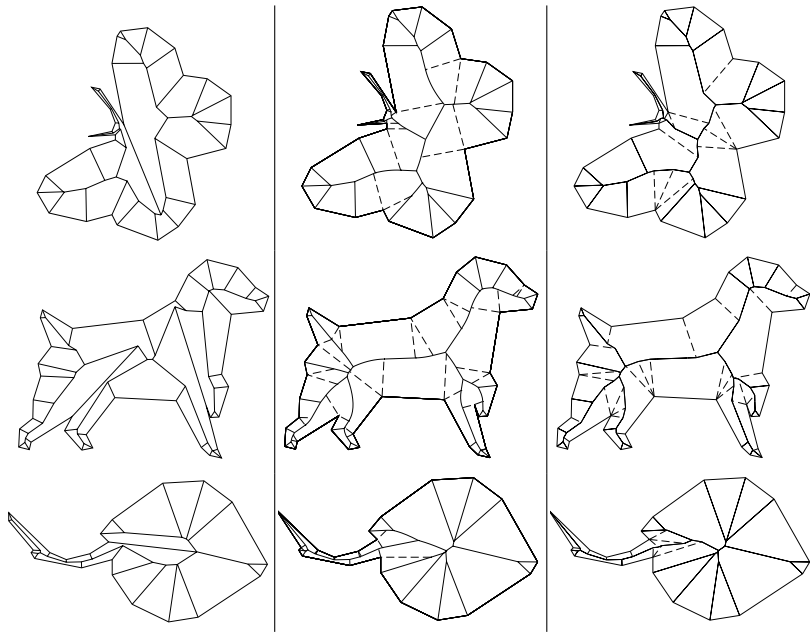
Once we have a sequence  $\kappa$  that ensures the  $\varepsilon$ -equivalence between the corresponding linear axis and the medial axis, we can construct this linear axis. The medial axis can be computed in linear time [9]. Despite its similarity to the medial axis, the fastest known algorithms for the straight skeleton computation are slower. The first sub-quadratic algorithm was proposed by Eppstein and Erickson [10]. It runs in  $O(n^{1+\epsilon} + n^{8/11+\epsilon}r^{9/11+\epsilon})$  time with a similar space complexity, where  $r$  is the number of reflex vertices and  $\epsilon$  is an arbitrarily small positive constant. A more recent algorithm by Cheng and Vigneron [11] computes the straight skeleton of a non-degenerate simple polygon in  $O(n \log^2 n + r\sqrt{r} \log r)$  expected time. For a degenerate simple polygon, its expected time bound is  $O(n \log^2 n + r^{17/11+\epsilon})$ . Any of these algorithms can be used to compute the straight skeleton of  $\mathcal{P}^\kappa(0)$ , where  $\mathcal{P}^\kappa(0)$  is the polygon obtained from  $P$  by inserting  $k_j$  zero-length edges at each reflex vertex  $v_j$ . The linear axis  $L^\kappa(P)$  corresponding to the sequence  $\kappa$  is then obtained from  $SS(\mathcal{P}^\kappa(0))$  by removing the bisectors incident to the reflex vertices of  $P$ .

However, if  $M(P)$  has only  $\varepsilon$ -clusters of constant size,  $L^\kappa(P)$  can be computed from the medial axis in linear time by adjusting the medial axis. In computing the linear axis, we adjust each non- $\varepsilon$ -edge of the medial axis to its counterpart in the linear axis. When adjusting an edge  $b_i b_j$  we first adjust the location of its endpoints to the location of the endpoints of its counterpart. If node  $b_i$  is part of an  $\varepsilon$ -cluster, we compute first the counterparts of the nodes in this cluster based on a local reconstruction of the linear wavefront. The adjustment of a node's location is done in constant time, if its  $\varepsilon$ -cluster has constant size. Finally, we use lemmas 3 and 4 to replace the parabolic arc or the perpendicular bisector with the corresponding chain of segments. We can now conclude that:

**Theorem 2.** *For a polygon with  $\varepsilon$ -clusters of constant size only, a linear axis  $\varepsilon$ -equivalent to the medial axis can be computed in linear time.*

## 5 Examples

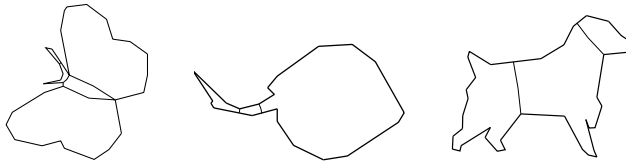
We have implemented the algorithm `ComputeHiddenEdges` of section 4.1 and the algorithm that constructs the linear axis from the medial axis described in section 4.2. Figure 5 illustrates the straight skeleton (left column), medial axis (middle column) and the linear axis (right column) of the contours of a butterfly, a dog and a ray. The contours come from the MPEG-7 Core Experiment test set “CE-Shape-1”, which contains images of white objects on a black background. The outer closed contour of the object in the image was extracted. In



**Fig. 5.** A comparison of the linear axis (right) with the medial axis (middle) and the straight skeleton (left). The dashed lines in the middle column are those Voronoi edges which are not part of the medial axis. The dashed lines in the right column represent the bisectors traced by the reflex vertices of the wavefront, which are not part of the linear axis. In these examples, the linear axis is isomorphic with the medial axis ( $\varepsilon = 0$ ).

this contour, each pixel corresponds to a vertex. The number of vertices were then decreased by applying the Douglas-Peucker [12] polygon approximation algorithm. For the medial axis computation we used the AVD LEDA package [13], which implements the construction of abstract Voronoi diagrams. The straight skeleton implementation was based on the straightforward algorithm in [14].

For the linear axes in figure 5, the number of hidden edges associated with a reflex vertex is indicated by the number of dashed-line bisectors incident to the reflex vertices that are not part of the linear axis. The difference between the number of incident bisectors and the number of hidden edges is one. We see that a very small number of hidden edges gives a linear axis  $\varepsilon$ -equivalent to the medial axis. The counter-intuitive results of the straight skeleton computation for the butterfly and dog are caused by sharp reflex vertices. Only two hidden edges for the reflex vertex between the dog's front legs and for the sharp reflex vertex in the butterfly, are sufficient to get a linear axis  $\varepsilon$ -equivalent to the medial axis. Though the reflex vertices of the ray contour are not very sharp, its straight skeleton indicates why this skeleton is unsuitable as shape descriptor. Two hidden edges associated to each reflex vertex at the body-tail junction, give a linear axis  $\varepsilon$ -equivalent to the medial axis. The largest number of hidden edges in these examples is three, for the reflex vertex between the dog's hind legs.



**Fig. 6.** Decompositions based on split events of the linear axis gives natural results even if the polygon contains sharp reflex vertices.

Figure 6 shows the results of the decomposition of the same contours as in figure 1, but this time based on the split events in the linear axis. We see that the unwanted effects of the sharp reflex vertices are eliminated and the results of this decomposition look more natural.

## 6 Concluding Remarks

The insertion of zero-length edges at reflex vertices decreases the speed of the wavefront points, which remedies the counter-intuitive effect of sharp reflex vertices in the straight skeleton. Another way to handle sharp reflex vertices could be to allow edges to translate in a non-parallel manner such that the variations in the speed of the wavefronts points are small along the wavefront. It is unclear however how to do this without increasing the complexity of the skeleton, or even if it is possible to do this in such a way that the outcome is a linear skeleton. Yet another way to take care of sharp reflex vertices is to apply first an outwards propagation, until the sharp reflex vertices have disappeared and then to propagate the front inwards. It is unclear however when to stop the outwards propagation such that relevant reflex vertices are not lost from the wavefront.

**Acknowledgements.** This research was partially supported by the Dutch Science Foundation (NWO) under grant 612.061.006, and by the FW6 IST Network of Excellence 506766 Aim@Shape.

## References

1. Blum, H.: A Transformation for Extracting New Descriptors of Shape. Symp. Models for Speech and Visual Form. ed: W. Wathen-Dunn. MIT Press (1967) 362–381.
2. Brady, M., Asada, H.: Smoothed Local Symmetries and their Implementation. *The International Journal of Robotics Research* **3**(3) (1984) 36–61.
3. Leyton, M.: A Process Grammar for Shape. *Art. Intelligence* **34** (1988) 213–247.
4. Ogniewicz, R. L., Kubler, O.: Hierarchic Voronoi Skeletons. *Pattern Recognition* **28**(3) (1995) 343–359.
5. Aichholzer, O., Aurenhammer, F.: Straight Skeletons for General Polygonal Figures in the Plane. In: *Proc. 2nd International Computing and Combinatorics Conference COCOON '96*. LNCS Vol. 1090. Springer-Verlag (1996) 117–126.

6. Aichholzer, O., Aurenhammer, F., Alberts, D., Gärtner, B.: A Novel Type of Skeleton for Polygons. *The Journal of Universal Computer Science*. **1** (1995) 752–761.
7. Tănase, M., Veltkamp, R.C.: Polygon Decomposition based on the Straight Line Skeleton. In: *Proc. 19th ACM Symp. on Computational Geometry*. (2003) 58–67.
8. Tănase, M., Veltkamp, R.C.: A Straight Skeleton Approximating the Linear Axis. TR (2004).
9. Chin, F., Snoeyink, J., Wang, C.-A.: Finding the Medial Axis of a Simple Polygon in Linear Time. *Discrete Computational Geometry* **21(3)** (1999) 405–420.
10. Eppstein, D., Erickson, J.: Raising Roofs, Crashing Cycles, and Playing Pool: Applications of a Data Structure for Finding Pairwise Interactions. *Discrete and Computational Geometry* **22(4)** (1999) 569–592.
11. Cheng, S.-W., Vigneron, A: Motorcycle Graphs and Straight Skeletons. In: *Proc. 13th ACM-SIAM Symp. Discrete Algorithms* (2002) 156–165.
12. Douglas, D.H., Peucker, T.K.: Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. *The Canadian Cartographer* **10(2)** (1973) 112–122.
13. AVD LEP, the Abstract Voronoi Diagram LEDA Extension Package.  
<http://www.mpi-sb.mpg.de/LEDA/friends/avd.html>.
14. Felkel, P., Obdržálek, Š.: Straight Skeleton Implementation. In: *Proc. of Spring Conference on Computer Graphics, Budmerice, Slovakia* (1998) 210–218.

# Non-additive Shortest Paths\*

George Tsaggouris<sup>1,2</sup> and Christos Zaroliagis<sup>1,2</sup>

<sup>1</sup> Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece

<sup>2</sup> Department of Computer Engineering and Informatics,  
University of Patras, 26500 Patras, Greece  
{tsaggour,zaro}@ceid.upatras.gr

**Abstract.** The non-additive shortest path (NASP) problem asks for finding an optimal path that minimizes a certain multi-attribute non-linear cost function. In this paper, we consider the case of a non-linear convex and non-decreasing function on two attributes. We present an efficient polynomial algorithm for solving a Lagrangian relaxation of NASP. We also present an exact algorithm that is based on new heuristics we introduce here, and conduct a comparative experimental study with synthetic and real-world data that demonstrates the quality of our approach.

## 1 Introduction

The shortest path problem is a fundamental problem in network optimization encompassing a vast number of applications. Given a digraph with a cost function on its edges, the shortest path problem asks for finding a path between two nodes  $s$  and  $t$  of total minimum cost. The problem has been heavily studied under the *additivity assumption* stating that the cost of a path is the sum of the costs of its edges. In certain applications, however, the cost function may not be additive along paths and may not depend on a single attribute. This gives rise to the so-called *non-additive shortest path* (NASP) problem, which asks for finding an optimal path that minimizes a certain multi-attribute *non-linear cost function*.

In this work, we are interested in the version of NASP in which every edge (and hence every path) is associated with two attributes, say cost and resource, and in which the non-linear objective function is convex and non-decreasing. This version of NASP has numerous applications in bicriteria network optimization problems, and especially in transportation networks; for example, it is the core subproblem in finding traffic equilibria [4,12]. In such applications, a utility cost function for a path  $p$  is defined that typically translates travel time (and travel cost if necessary) to a common utility cost measure (e.g., money). Experience shows that users of traffic networks value time non-linearly [8]: small amounts of time have relatively low value, while large amounts of time are very valuable. Consequently, the most interesting theoretical models for traffic equilibria [4,12] involve minimizing a monotonic non-linear utility function of two attributes, travel cost and travel time, that allows a decision

---

\* This work was partially supported by the IST Programme (6th FP) of EC under contract No. IST-2002-001907 (integrated project DELIS).

maker to find the optimal path both w.r.t. travel cost and time. NASP is related to multiobjective optimization, and actually appears to be a core problem in this area; e.g., in multi-agent competitive network models [5]. The goal in multiobjective optimization problems is to compute a set of solutions (paths in the case of NASP) optimizing several cost functions (criteria). These problems are typically solved by producing the so-called *Pareto curve* [11], which is the set of all undominated Pareto-optimal solutions or “trade-offs” (the set of feasible solutions where the attribute-vector of one solution is not dominated by the attribute-vector of another solution). Multiobjective optimization problems are usually NP-hard. The difficulty in solving them stems from the fact that the Pareto curve is typically exponential in size, and this applies to NASP itself. Note that even if a decision maker is armed with the entire set of all undominated Pareto-optimal solutions, s/he is still left with the problem of which is the “best” solution for the application at hand. Consequently, three natural approaches to solve multiobjective optimization problems are: (i) optimize one objective while bounding the rest (cf. e.g., [2,6,9]); (ii) find approximation schemes that generate a polynomial number of solutions which are within a small factor in all objectives [11]; (iii) proceed in a normative way and choose the “best” solution by introducing a utility function on the attributes involved.

In this paper, we concentrate on devising an exact algorithm for solving NASP as well as on computing a Lagrangian relaxation of the integer non-linear programming (INLP) formulation of the problem, which is the first step in finding the optimal solution. Solving the Lagrangian relaxation of NASP is equivalent to computing the “best” path among those that lie on the convex hull of the Pareto curve, called extreme paths. Note that these paths may also be exponentially many and that the best extreme path is not necessarily the optimal [10]. To find the best extreme path, we adopt the so-called hull approach, which is a cutting plane method that has been used in various forms and by various researchers since 1966 [3]. Its first explicit use in bicriteria network optimization was made in [6] for solving the Resource Constrained Shortest Path (RCSP)<sup>1</sup> problem. However, only recently a considerable refinement of the method has been made as well as its worst-case complexity was carefully analyzed [9]. Variations of the hull approach for solving relaxations of NASP, when the non-linear objective function is convex and non-decreasing, have previously appeared in [7,10], but with very crude time analyses giving bounds proportional to the number of the extreme Pareto paths. Regarding the exact solution to NASP, we adopt and extend the classical 3-phase approach introduced in [6] for solving RCSP, used later in [2], and considerably refined in [9]: (a) compute lower and upper bounds to the optimal solution, using the solution of the relaxed problem; (b) prune the graph by deleting nodes and edges that cannot belong to an optimal path; (c) close the gap between upper and lower bound. The pruning phase was not considered in [6, 7]. Other approaches for solving NASP, either involved heuristics based on path flow formulations and repeatedly solving LPs [4,5], or were based on repeated applications of RCSP [12] (one for each possible value of the resource of a path).

---

<sup>1</sup> RCSP asks for finding a minimum cost path whose resource does not exceed a specific bound. Costs and resources are *additive* along paths. RCSP is modelled as an ILP.



In this work, we provide new theoretical and experimental results for solving NASP. Our contributions are threefold: (i) We extend the hull algorithm in [9] and provide an efficient polynomial time algorithm for solving the Lagrangian relaxation of NASP. Note that the extension of a method that solves the relaxation of a linear program (RCSP) to a method that solves the relaxation of a non-linear program (NASP) is not straightforward. For integral costs and resources in  $[0, C]$  and  $[0, R]$ , respectively, our algorithm runs in  $O(\log(nRC)(m + n \log n))$  time, where  $n$  and  $m$  are the number of nodes and edges of the graph. To the best of our knowledge, this is the first efficient algorithm for solving a relaxation of NASP. Our algorithm provides both upper and lower bounds, while the one in [10] provides only the former. (ii) We give an exact algorithm to find the optimum of NASP based on the aforementioned 3-phase approach. Except for the relaxation method, our algorithm differentiates from those in [6,7,9,10] also in the following: (a) we provide two new heuristics, the *gradient* and the *on-line node reductions* heuristics, that can be plugged in the extended hull algorithm and considerably speed up the first phase as our experimental study shows. (b) We give a method for gap closing that is not based on  $k$ -shortest paths, as those in [6,7], but constitutes an extension and improvement of the method for gap closing given in [9] for the RCSP problem. Again, our experiments show that our method is faster than those in [6,7,9]. (iii) We introduce a generalization of both NASP and RCSP, in which one wishes to minimize a non-linear, convex and non-decreasing function of two attributes such that both attributes do not exceed some specific bounds, and show that the above results hold for the generalized problem, too.

Proofs and parts omitted due to space limitations can be found in [13].

## 2 Preliminaries and Problem Formulation

In the following, let  $G = (V, E)$  be a digraph with  $n$  nodes and  $m$  edges, and let  $c : E \rightarrow \mathbb{R}_0^+$  be a function associating non-negative costs to the edges of  $G$ . The cost of an edge  $e \in E$  will be denoted by  $c(e)$ . To better introduce the definition of NASP, we start with an Integer Linear Programming (ILP) formulation of the well-known additive shortest path problem given in [9]. Let  $P$  be the set of paths from node  $s$  to node  $t$  in  $G$ . For each path  $p \in P$  we introduce a 0-1 variable  $x_p$ , and we denote by  $c_p$ , or by  $c(p)$ , the path's cost, i.e.,  $c_p \equiv c(p) = \sum_{e \in p} c(e)$ . Then, the ILP for the additive shortest path is as follows.

$$\min \sum_{p \in P} c_p x_p \quad (1)$$

$$s.t. \sum_{p \in P} x_p = 1 \quad ; \quad x_p \in \{0, 1\}, \forall p \in P \quad (2)$$

Suppose now that, in addition to the cost function  $c$ , there is also a resource function  $r : E \rightarrow \mathbb{R}_0^+$  associating each edge  $e \in E$  with a non-negative resource  $r(e)$ . We denote the resource of a path  $p$  by  $r_p$ , or by  $r(p)$ , and define it as  $r_p \equiv r(p) = \sum_{e \in p} r(e)$ . Consider now an non-decreasing and convex function

$U : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  with first derivative  $U' : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ . Function  $U$  is a translation function that converts the resource of a path to cost. Now, the shortest path problem asks for finding a path  $p \in P$  which minimizes the overall cost, that is:

$$\min \sum_{p \in P} c_p x_p + U\left(\sum_{p \in P} r_p x_p\right) \quad (3)$$

Substituting (1) with the new objective function (3) and keeping the same constraints (2), we get an Integer Non-Linear Program (INLP). Since the overall cost of a path is no longer additive on its edges the problem is called the *non-additive shortest path problem* (NASP). It is easy to see that the solution to NASP is a Pareto-optimal path. In the full generality of the problem, there is also a translation function for cost (cf. Section 5). We start from this simpler objective function, since it is the most commonly used in applications (see e.g., [4,5,12]), and it better introduces our approach.

### 3 Lagrangian Relaxation and Its Solution

We can transform the above INLP into an equivalent one by introducing a variable  $z$  and setting it equal to  $z = \sum_{p \in P} r_p x_p$ . Thus, the INLP objective function becomes  $\min \sum_{p \in P} c_p x_p + U(z)$ , while an additional equality constraint is introduced:  $\sum_{p \in P} r_p x_p = z$ . Since  $U(\cdot)$  is a non-decreasing function, we can relax this equality constraint to inequality, and get the following INLP:

$$\begin{aligned} \min \quad & \sum_{p \in P} c_p x_p + U(z) \\ \text{s.t.} \quad & \sum_{p \in P} r_p x_p \leq z \\ & \sum_{p \in P} x_p = 1 \quad ; \quad x_p \in \{0, 1\}, \forall p \in P \end{aligned} \quad (4)$$

We can relax the above using Lagrangian relaxation: introduce a Lagrangian multiplier  $\mu \geq 0$ , multiply the two sides of (4) by  $\mu$ , and turn constraint (4) in the objective function. The objective now becomes  $L(\mu) = \min \sum_{p \in P} (c_p + \mu r_p) x_p + U(z) - \mu z$ , while constraints (2) remain unchanged.  $L(\mu)$  is a lower bound on the optimal value of the objective function (3) for any value of the Lagrangian multiplier  $\mu$  [1, Ch. 16]. A best lower bound can be found by solving the problem  $\max_{\mu \geq 0} L(\mu)$  (equivalent to solving the *NLP relaxation* of NASP, obtained by relaxing the integrality constraints). Since none of the constraints in  $L(\mu)$  contains variable  $z$ , the problem decomposes in two separate problems  $L_p(\mu)$  and  $L_z(\mu)$  such that

$$L(\mu) = L_p(\mu) + L_z(\mu),$$

$$\begin{aligned}
\text{where} \quad L_p(\mu) &= \min \sum_{p \in P} (c_p + \mu r_p) x_p \\
&\quad \text{s.t.} \quad \sum_{p \in P} x_p = 1 \quad ; \quad x_p \in \{0, 1\}, \forall p \in P \\
\text{and} \quad L_z(\mu) &= \min U(z) - \mu z
\end{aligned} \tag{5}$$

Observe now that for a fixed  $\mu$ : (i)  $L_p(\mu)$  is a standard additive shortest path problem w.r.t. the composite (non-negative) edge weights  $w(\cdot) = c(\cdot) + \mu r(\cdot)$  and can be solved using Dijkstra's shortest path algorithm in  $O(m + n \log n)$  time; (ii)  $L_z(\mu)$  is a standard minimization problem of a (single variable) convex function. These crucial observations along with some technical lemmata will allow us to provide an efficient polynomial algorithm for solving the relaxed version (Lagrangian relaxation) of the above INLP.

### 3.1 The Ingredients

In the following, let  $p = SP(s, t, c, r, \mu)$  denote a shortest path routine that w.l.o.g. always returns the same  $s$ - $t$  path  $p$  that is shortest w.r.t. edge weights  $w(e) = c(e) + \mu r(e)$ ,  $\forall e \in E$ , and let  $\arg \min\{f(z)\}$  denote the maximum  $z$  that minimizes  $f(z)$ . The next lemma relates resources and costs of shortest paths w.r.t. different  $\mu$ .

**Lemma 1.** *Let  $p_l = SP(s, t, c, r, \mu_l)$ ,  $p_h = SP(s, t, c, r, \mu_h)$ ,  $z_l = \arg \min\{U(z) - \mu_l z\}$ ,  $z_h = \arg \min\{U(z) - \mu_h z\}$ , and  $0 \leq \mu_l \leq \mu_h$ . Then, (i)  $r(p_l) \geq r(p_h)$ ; (ii)  $c(p_l) \leq c(p_h)$ ; (iii)  $r(p_l) - z_l \geq r(p_h) - z_h$ ; and (iv)  $c(p_l) + U(z_l) \leq c(p_h) + U(z_h)$ .*

Lemma 1 allows us to use any line-search method to either find an optimal solution or a lower bound to the problem. Here, we follow the framework of the hull approach and in particular of its refinement in [9] for solving the RCSP problem (hull algorithm), and extend it to NASP. The approach can be viewed as performing a kind of binary search between a feasible and an infeasible solution w.r.t. constraint (4). In each step the goal is to find a pair  $(p, z)$  that either corresponds to a new feasible solution that reduces the objective value ( $c(p) + U(z)$ ), or corresponds to an infeasible solution that decreases the amount ( $r(p) - z$ ) by which constraint (4) is violated.

The next two lemmata allow the effective extension of the hull algorithm to NASP. The first lemma shows how such a solution can be found by suitably choosing the Lagrangian multiplier at each iteration.

**Lemma 2.** *Let  $p_l = SP(s, t, c, r, \mu_l)$ ,  $p_h = SP(s, t, c, r, \mu_h)$ ,  $z_l = \arg \min\{U(z) - \mu_l z\}$ ,  $z_h = \arg \min\{U(z) - \mu_h z\}$ ,  $p_m = SP(s, t, c, r, \mu_m)$ ,  $z_m = \arg \min\{U(z) - \mu_m z\}$ ,  $0 \leq \mu_l \leq \mu_h$ , and  $\mu_m = \frac{c(p_h) - c(p_l)}{r(p_l) - r(p_h)}$ . Then, (i)  $r(p_l) - z_l \geq r(p_m) - z_m \geq r(p_h) - z_h$ , and (ii)  $c(p_l) + U(z_l) \leq c(p_m) + U(z_m) \leq c(p_h) + U(z_h)$ .*

The previous lemma can be usefully applied when we have an efficient way to test for feasibility. Compared to RCSP, where this test was trivial (just checking

whether the resource of a path is smaller than a given bound), checking whether  $(p, z)$  is feasible reduces in finding the inverse of  $U'(z)$  (as objective (5) dictates) and this may not be trivial. The next lemma allows us not only to efficiently test for feasibility, but also to avoid finding the inverse of  $U'(z)$ .

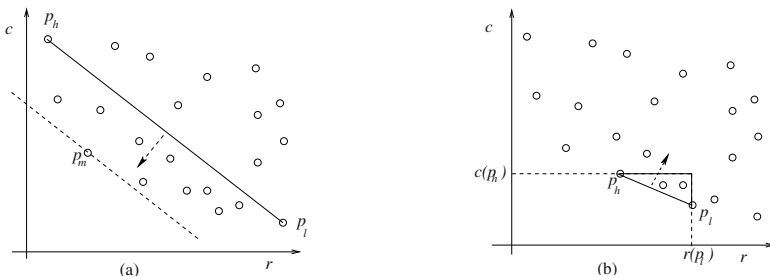
**Lemma 3.** *Let  $\mu \geq 0$  with corresponding path  $p = SP(s, t, c, r, \mu)$ , and let  $z^* = \arg \min\{U(z) - \mu z\}$ . Then,  $(p, z^*)$  is a feasible solution w.r.t. constraint (4) if  $U'(r(p)) \leq \mu$ .*

### 3.2 The Extended Hull Algorithm

We are now ready to give the algorithm for solving the relaxed INLP. Call a multiplier  $\mu$  *feasible* if  $\mu$  leads to a feasible solution pair  $(p, z)$ ; otherwise, call  $\mu$  *infeasible*. During the process we maintain two multipliers  $\mu_l$  and  $\mu_h$  as well as the corresponding paths  $p_l$  and  $p_h$ . Multiplier  $\mu_l$  is an infeasible one, while  $\mu_h$  is feasible. Initially,  $\mu_l = 0$  and the corresponding path  $p_l$  is the minimum cost path, while  $\mu_h = +\infty$  and the corresponding path  $p_h$  is the minimum resource path. We adopt the geometric interpretation introduced in [9] and represent each path as a point in the  $r$ - $c$ -plane.

In each iteration of the algorithm we find a path  $p_m$  corresponding to the multiplier  $\mu_m = \frac{c(p_h) - c(p_l)}{r(p_l) - r(p_h)}$ . This is equivalent to moving the line connecting  $p_h$  and  $p_l$  with slope  $-\mu_m$  along its normal direction until we hit an extreme point in that direction (see Fig. 1(a)). Multiplier  $\mu_m$  leads to a solution  $(p_m, z_m)$ . We check whether  $(p_m, z_m)$  is feasible using Lemma 3. This is equivalent to check whether  $U'(r(p_m)) - \mu_m$  is positive, negative, or zero. In the latter case, we have found the optimal solution. If it is positive, then by Lemma 2 we have to move to the “left” by setting  $p_l$  equal to  $p_m$  and  $\mu_l$  to  $\mu_m$ ; otherwise, we have to move to the “right” by setting  $p_h$  equal to  $p_m$  and  $\mu_h$  to  $\mu_m$ .

We iterate this procedure as long as  $\mu_l < \mu_m < \mu_h$ , i.e., as long as we are able to find a new path below the line connecting the paths  $p_l$  and  $p_h$ . When such a path cannot be found, the algorithm stops. As we shall see next, in this case either one of  $p_l$  and  $p_h$  is optimal, or there is a duality gap. We call the above the *extended hull algorithm*.



**Fig. 1.** The algorithm. (a) First iteration. (b) Last iteration.

We are now ready to discuss its correctness. The following lemma implies that the paths  $p_l$  and  $p_h$  maintained during each iteration of the algorithm give us upper bounds on the resource and the cost of the optimal path.

**Lemma 4.** *Let  $\mu \geq 0$  be a fixed multiplier and let  $p = SP(s, t, c, r, \mu)$  be its corresponding path. (i) If  $U'(r(p)) > \mu$ , then  $r(p)$  is an upper bound on the resource of the optimal path. (ii) If  $U'(r(p)) < \mu$ , then  $c(p)$  is an upper bound on the cost of the optimal path. (iii) If  $U'(r(p)) = \mu$ , then  $p$  is optimal.*

The next lemma (see Fig. 1(b)) completes the discussion on the correctness.

**Lemma 5.** *Let  $0 < \mu_l < \mu_h$  be the multipliers at the end of the algorithm,  $p_l = SP(s, t, c, r, \mu_l)$  and  $p_h = SP(s, t, c, r, \mu_h)$  be their corresponding paths, and let  $\mu_m = \frac{c(p_h) - c(p_l)}{r(p_l) - r(p_h)}$ . (i) If  $\mu_l \leq U'(r(p_l)) \leq \mu_m$ , then  $p_l$  is optimal. (ii) If  $\mu_m \leq U'(r(p_h)) \leq \mu_h$ , then  $p_h$  is optimal. (iii) If  $U'(r(p_h)) < \mu_m < U'(r(p_l))$ , then there may be a duality gap. In this case, the optimum lies in the triangle defined by the lines  $r = r(p_l)$ ,  $c = c(p_h)$ , and the line connecting  $p_l$  and  $p_h$ .*

*Proof.* From Lemma 2, we have that  $\mu_l \leq \mu_m = \frac{c(p_h) - c(p_l)}{r(p_l) - r(p_h)} \leq \mu_h$ . We will also use the fact (cf. [13]) that if we have found a path which is shortest for both multipliers  $\mu_l$  and  $\mu_h$ , then it is shortest for any  $\mu \in [\mu_l, \mu_h]$ .

(i) If  $\mu_l \leq U'(r(p_l)) \leq \mu_m$ , from the above fact we have that  $\forall q \in P$ :  $c(p_l) + U'(r(p_l))r(p_l) \leq c(q) + U'(r(p_l))r(q)$ . Thus, Lemma 4(iii) implies that  $p_l$  is optimal. (ii) Similar to (i). (iii) Follows directly from Lemma 4 and the stopping condition of the algorithm.  $\square$

In the case where there is a duality gap (see Fig. 1(b)), we can compute a lower bound to the actual optimum of the original INLP. Let the line connecting  $p_l$  and  $p_h$  be  $c + \mu r = W$ , where  $W = c(p_l) + \mu r(p_l) = c(p_h) + \mu r(p_h)$ . In the best case the optimum path  $(\bar{c}, \bar{r})$  lies on that line and therefore it must be  $\bar{c} + \mu \bar{r} = W$ . A lower bound to the actual optimum is thus  $\bar{c} + U(\bar{r})$ , where  $(\bar{c}, \bar{r}) = (W - \mu[U']^{-1}(\mu), [U']^{-1}(\mu))$  is the solution of the system of equations  $\bar{c} + \mu \bar{r} = W$  and  $U'(\bar{r}) = \mu$  (here  $[U']^{-1}(\cdot)$  denotes the inverse function of  $U'(\cdot)$ ).

We now turn to the running time of the algorithm. The algorithm's progress can be measured by how much the area of the triangular unexplored region (defined by points  $p_l$ ,  $p_h$ , and the intersection of the lines with slopes  $-\mu_l$  and  $-\mu_h$  passing through  $p_l$  and  $p_h$ , respectively) reduces at each iteration. We can prove (in a way similar to that in [9]; cf. [13]) that the area reduces by at least  $3/4$ . For integral costs and resources in  $[0, C]$  and  $[0, R]$ , respectively, the area of the initial triangle is at most  $\frac{1}{2}(nR)(nC)$ , the area of the minimum triangle is  $\frac{1}{2}$ , and the unexplored region is at least divided by four at each iteration. Thus, the algorithm ends after  $O(\log(nRC))$  iterations. In each iteration a call to Dijkstra's algorithm is made. Hence, we have established the following.

**Theorem 1.** *The extended hull algorithm solves the Lagrangian relaxation of NASP in  $O(\log(nRC)(m + n \log n))$  time.*

## 4 The Exact Algorithm

The exact algorithm for solving NASP follows the general outline of the algorithms in [2,6,9] for solving RCSP and consists of three phases. In the first phase, we solve the relaxed problem using the extended hull algorithm. This provides us with a lower and an upper bound for the objective value as well as with an upper bound on the cost and the resource of the optimal path. In the second phase, we use these cost and resource upper bounds to prune the graph by deleting nodes and edges that cannot lie on the optimal path (details for this phase can be found in [13]). In the third phase, we close the gap between lower and upper bound and find the actual optimum by applying a path enumerating process in the pruned graph.

### 4.1 Improving the Efficiency of the Extended Hull Algorithm

To achieve better running times for phase 1, we develop two new heuristics that can be plugged in the extended hull algorithm. The first heuristic intends to reduce the number of iterations, accomplished by identifying cases where we can increase the algorithm's progress by selecting a multiplier other than that suggested by the hull approach. We refer to it as the *gradient* heuristic. The second heuristic aims at reducing the problem size by performing node reductions on the fly, using information obtained from previous iterations. We refer to it as the *on-line node reductions* heuristic.

**Gradient Heuristic.** Assume that there exists a multiplier  $\mu'$  such that  $\mu_l < \mu' < \mu_h$ , and for which we can tell a priori whether the corresponding path is feasible or not (i.e., without computing it). We can then distinguish two cases in which we can examine  $\mu'$  instead of  $\mu_m$ , and achieve a better progress: (a) If  $\mu'$  leads to an infeasible solution and  $\mu_l < \mu_m < \mu'$ , then Lemma 1 implies that  $\mu_m$  also leads to an infeasible solution. Therefore, since both  $\mu_m$  and  $\mu'$  lead to an infeasible path, and  $\mu'$  is closer to  $\mu_h$  than  $\mu_m$ , we can use  $\mu'$  instead of  $\mu_m$  in the update step and thus make a greater progress. (b) Similarly, if  $\mu'$  leads to a feasible solution and  $\mu' < \mu_m < \mu_h$ , then  $\mu_m$  also leads to a feasible solution. Again, we can use  $\mu'$  instead of  $\mu_m$  in the update step, since it is closer to  $\mu_l$ . The following lemma suggests that when we are given a feasible (resp. infeasible) multiplier  $\mu$ , we can always find an infeasible (resp. feasible) multiplier  $\mu'$ . Its proof follows by Lemma 1 and the convexity of  $U(\cdot)$ .

**Lemma 6.** Let  $\mu \geq 0$  be a fixed multiplier,  $p = SP(s, t, c, r, \mu)$ ,  $\mu' = U'(r(p))$ , and  $q = SP(s, t, c, r, \mu')$ . (i) If  $\mu' \leq \mu$ , then  $U'(r(q)) \geq \mu'$ . (ii) If  $\mu' \geq \mu$ , then  $U'(r(q)) \leq \mu'$ .

Lemma 6 along with the above discussion lead to the following heuristic that can be plugged in the extended hull algorithm. Let  $0 \leq \mu_l \leq \mu_h$  be the multipliers at some iteration of the algorithm, and let  $p_l, p_h$  be the corresponding paths. If  $U'(r(p_h)) \geq \frac{c(p_h) - c(p_l)}{r(p_l) - r(p_h)} \geq \mu_l$ , then set  $\mu_m = U'(r(p_h))$ . If  $U'(r(p_l)) \leq \frac{c(p_h) - c(p_l)}{r(p_l) - r(p_h)} \leq \mu_h$ , then set  $\mu_m = U'(r(p_l))$ . Otherwise, set  $\mu_m = \frac{c(p_h) - c(p_l)}{r(p_l) - r(p_h)}$ .

**On-line Node Reductions Heuristic.** Further progress to the extended hull algorithm can be made by temporarily eliminating at each iteration those nodes that cannot lie on any path obtained by any subsequent iteration of the algorithm. These temporary eliminations of nodes are done in an on-line fashion and apply only to phase 1. Let  $d_\mu(x, y)$  denote a lower bound on the cost of a shortest  $x$ - $y$  path w.r.t. the edge weights  $w(\cdot) = c(\cdot) + \mu r(\cdot)$ . Then, we can safely exclude from the graph any node  $u$  for which any of the following holds:

$$d_{\mu_m}(s, u) + d_{\mu_m}(u, t) > c(p_h) + \mu_m r(p_h) \quad (6)$$

$$d_{\mu_l}(s, u) + d_{\mu_l}(u, t) > c(p_h) + \mu_l r(p_h) \quad (7)$$

$$d_{\mu_h}(s, u) + d_{\mu_h}(u, t) > c(p_l) + \mu_h r(p_l) \quad (8)$$

The above lower bound distances can be found using the following lemma.

**Lemma 7.** *Let  $d_{\mu_l}(u, v)$  and  $d_{\mu_h}(u, v)$  be lower bounds of the cost of a shortest  $u$ - $v$  path w.r.t. edge weights  $c(\cdot) + \mu_l r(\cdot)$  and  $c(\cdot) + \mu_h r(\cdot)$ , respectively, where  $\mu_l < \mu_h$ . Let  $\bar{\mu} \in [\mu_l, \mu_h]$ . Then,  $d_{\bar{\mu}}(u, v) = \theta d_{\mu_l}(u, v) + (1 - \theta) d_{\mu_h}(u, v)$  is also a lower bound of the cost of a shortest  $u$ - $v$  path w.r.t. edge weights  $c(\cdot) + \bar{\mu} r(\cdot)$ , where  $\theta = \frac{\bar{\mu}_h - \bar{\mu}}{\bar{\mu}_h - \bar{\mu}_l}$ .*

To apply the on-line node reductions heuristic, we maintain a set  $R$  that consists of the nodes reduced until that iteration. Additionally, we maintain the lower bound distances  $d_{\mu_l^s}(s, u)$ ,  $d_{\mu_h^s}(s, u)$ ,  $d_{\mu_l^t}(u, t)$ ,  $d_{\mu_h^t}(u, t)$ ,  $\forall u \in V - R$ , where  $\mu_l^s \leq \mu_l$ ,  $\mu_l^t \leq \mu_l$ ,  $\mu_h^s \geq \mu_h$ ,  $\mu_h^t \geq \mu_h$ . These distances have been computed in previous shortest path computations (initially, they are set to zero). We call  $u$  *reduced* either if  $u \in R$ , or if any of the inequalities (6), (7), or (8) is satisfied.

In each iteration we test a multiplier  $\mu_m$  by performing a forward (resp. backward) run of Dijkstra's algorithm [13], using  $w(\cdot) = c(\cdot) + \mu_m r(\cdot)$  as edge weights. For each node  $u$  extracted from the priority queue (used in Dijkstra's algorithm), we check whether  $u$  is reduced or not. If  $u$  is reduced but not in  $R$ , then we insert it to  $R$ . To check the inequalities, we need the lower bound distances in the LHS of (6), (7), and (8), which can be found by applying Lemma 7 on the known distances  $d_{\mu_l^s}(s, u)$ ,  $d_{\mu_h^s}(s, u)$ ,  $d_{\mu_l^t}(u, t)$ , and  $d_{\mu_h^t}(u, t)$ . If  $u$  is reduced, then we do not need to examine its outgoing (resp. incoming) edges. We stop Dijkstra's algorithm when the target (resp. source) node is settled. At this point, by setting the distances of the non-reduced unsettled nodes to the distance of the target (resp. source), we have new lower bound distance information, which can be used in subsequent iterations by updating the current information appropriately.

## 4.2 Closing the Gap

To close the gap, we follow a label setting method that is essentially a modification of the one proposed in [9] for the RCSP problem. In the additive shortest path problem, Dijkstra's algorithm keeps a label for each node  $u$  representing the tentative distance of an  $s$ - $u$  path. In our case each path  $p$  has a cost and a resource, and thus its corresponding label is a tuple  $(r_p, c_p)$ . Consequently, with each node  $u$  a set of labels corresponding to  $s$ - $u$  paths have to be stored, since now

there is no total order among them. However, we can exploit the fact that some paths may *dominate* other paths, where a path  $p$  with label  $(r_p, c_p)$  dominates path  $q$  with label  $(r_q, c_q)$  if  $c_p \leq c_q$  and  $r_p \leq r_q$ . Hence, at each node  $u$  we maintain a set  $L(u)$  of undominated  $s$ - $u$  paths, that is, every path in  $L(u)$  does not dominate any other in the set. Let  $c_{uv}$  (resp.  $r_{uv}$ ) denote the cost (resp. resource) of the shortest  $u$ - $v$  path w.r.t. cost (resp. resource), and let  $c_{max}$  (resp.  $r_{max}$ ) be the upper bound on the cost (resp. resource) of the optimal path returned by phase 1. We call an  $s$ - $u$  path  $p$  of cost  $c_p$  and resource  $r_p$  *non-promising*, if any of the following holds: (i)  $c_p + c_{ut} > c_{max}$ ; (ii)  $r_p + r_{ut} > r_{max}$ ; (iii) if there exists a Pareto-optimal  $u$ - $t$  path  $q$  of cost  $c_q$  and resource  $r_q$  for which  $c_p + c_q > c_{max}$  and  $r_p + r_q > r_{max}$ . Clearly, a non-promising path cannot be extended to an optimal one and hence it can be discarded. We would like to mention that in [9] only (i) and (ii) are checked. Our experiments revealed, however, that gap closing involving (iii) can be considerably faster (cf. Section 6). The gap closing algorithm is implemented with the use of a priority queue, whose elements are tuples of the form  $(w_p, (r_p, c_p))$ , where  $w_p = c_p + \mu_m r_p$  is the combined cost of  $p$  and  $\mu_m$  is the final multiplier of phase 1. Paths are examined in the order they are hit by sweeping the line connecting  $p_l$  and  $p_h$  along its normal direction.

## 5 Generalization

The generalization of the objective function (3) involves two non-decreasing and convex functions  $U_c : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  and  $U_r : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  translating a path's cost (travel distance) and resource (travel time), respectively, to the common utility cost (money). We can also incorporate restrictions that may exist on the total cost or the resource of a path, thus adding constraints of the form  $\sum_{p \in P} c_p x_p \leq B_c$  and  $\sum_{p \in P} r_p x_p \leq B_r$ , for some constants  $B_c, B_r \geq 0$ . Consequently, the resulting INLP is as follows

$$\begin{aligned} \min \quad & U_c \left( \sum_{p \in P} c_p x_p \right) + U_r \left( \sum_{p \in P} r_p x_p \right) \\ \text{s.t.} \quad & \sum_{p \in P} c_p x_p \leq B_c \quad ; \quad \sum_{p \in P} r_p x_p \leq B_r \\ & \sum_{p \in P} x_p = 1 \quad ; \quad x_p \in \{0, 1\}, \quad \forall p \in P \end{aligned} \tag{9}$$

and constitutes a generalization to both NASP and RCSP. We can show that the idea with the non-negative composite edge weights  $w(\cdot) = c(\cdot) + \mu r(\cdot)$  (based on a Lagrangian multiplier  $\mu$ ) provides us with a polynomial algorithm for solving the relaxed version of the above INLP. We provide generalized versions of the key Lemmata 4 and 5 (see [13]), which actually allow us to apply the hull approach and get the desired solution. The generalized versions of these Lemmata result in certain modifications of the extended hull algorithm (see [13]), without sacrificing its worst-case complexity.



## 6 Experiments

We conducted a series of experiments based on several implementations<sup>2</sup> of the extended hull algorithm and of the exact algorithm with the heuristics described in Section 4. In particular, for the solution of the Lagrangian relaxation of NASP (phase 1 of the exact algorithm), we have implemented the extended hull algorithm (EHA) of Section 3.2 (using Dijkstra's algorithm for shortest path computations and aborting when the target node is found), the variant of the hull approach described in [10] using the bidirectional Dijkstra's algorithm (MW), and EHA using different heuristics: EHA[B] (with the bidirectional Dijkstra's algorithm), EHA[G] (with the gradient heuristic), EHA[NR] (with the on-line node reductions heuristic), EHA[G+NR] (gradient and on-line node reductions heuristics), and EHA[G+B] (gradient heuristic with bidirectional Dijkstra).

With this bulk of implementations, we performed experiments on random grid graphs and digital elevation models (DEM) – following the experimental set-up in [9] – as well as on real-world data based on US road networks. For grid graphs, we used integral edge costs and resources chosen randomly and independently from  $[100, 200]$ . A DEM is a grid graph where each node has an associated height value. We used integral values as heights chosen randomly and independently from  $[100, 200]$ . The same applies for the edge resources. The cost of an edge was the absolute difference between the heights of its endpoints. In both types of input, we used the objective function (3) which is the most commonly used in transportation applications [4,5,12] with  $U(r)$  quadratic and appropriately scaled; that is, the objective function was  $\frac{c}{\delta_c(s,t)} + (\frac{r}{\delta_r(s,t)})^2$ , where by  $\delta_\lambda(s,t)$  we denote the weight of the shortest  $s$ - $t$  path with respect to the weight function  $\lambda(\cdot)$ .

The experimental results on grid graphs are reported in Table 1 (similar results hold for DEMs; cf. [13]). We report on time performances (secs) and on the number of shortest path computations (#SP). The reported values are averages over several query  $(s,t)$  pairs that involve border nodes of the grid (similar results hold for randomly chosen pairs).

For the solution of the relaxed problem (phase 1 of the exact algorithm), we observe that MW, despite the fact that it is implemented using bidirectional Dijkstra (which is usually faster than pure Dijkstra), has an inferior performance w.r.t. both EHA and EHA[B]. This is due to the fact that it performs much more shortest path computations, since MW does not use derivatives. Regarding the heuristics, we observe that both gradient and on-line node reductions are rather powerful heuristics. In all cases, the gradient heuristic reduces the number of shortest path computations from 20% to 60%, and the improvement is more evident as the graph size gets larger. The on-line node reductions heuristic achieves in almost all cases a better running time than the gradient heuristic. The most interesting characteristic of this heuristic is that its performance increases with the number of shortest path computations. This is due to the successive reduction of the problem size. Finally, the combination of both heuristics gives

<sup>2</sup> Implementations were in C++ using LEDA. Experiments were carried out in a SunFire 280R with an UltraSPARC-III processor at 750 MHz and 512 MB of memory.

**Table 1.** #GCP denotes the number of  $(s, t)$  pairs that needed pruning and gap closing.

$N \times N$ Grid – 100 Border $(s, t)$ pairs										
$N$	50		100		200		400		600	
Implem.	#SP	Time	#SP	Time	#SP	Time	#SP	Time	#SP	Time
MW	9.35	0.125	10.9	0.650	12.3	4.862	14.8	33.540	15.5	91.640
EHA	6.74	0.095	7.61	0.490	8.31	3.359	9.68	22.128	9.97	61.627
EHA[B]	6.74	0.088	7.61	0.444	8.31	3.211	9.68	21.648	9.97	57.902
EHA[G]	4.35	0.061	4.14	0.260	4.38	1.721	4.73	10.609	4.62	27.965
EHA[NR]	6.74	0.063	7.61	0.281	8.31	1.683	9.68	9.6354	9.97	25.354
EHA[G+NR]	4.35	0.053	4.14	0.234	4.38	1.451	4.73	8.3676	4.62	21.879
EHA[G+B]	4.35	0.054	4.14	0.233	4.38	1.650	4.73	10.437	4.62	26.290

$N \times N$ Grid – 10000 Border $(s, t)$ pairs					
$N$	100		200		
#GCP	101		182		
Phase	Time	#del_min	Time	#del_min	
pruning	0.267	–	2.209	–	
gc[MZ]	0.359	14260.9	32.420	744056	
gc[new]	0.276	9459.8	17.440	456400	

**Table 2.** Nodes  $s$  and  $t$  are chosen randomly among all nodes.

Road Networks – 1000 Random $(s, t)$ pairs							
Input Graph	U.S. NHPN		Florida		Alabama		
Implem.	#SP	Time	#SP	Time	#SP	Time	
MW	10.58	3.43	10.19	2.05	9.88	2.45	
EHA	7.34	2.44	7.10	1.33	6.88	2.14	
EHA[B]	7.34	2.32	7.10	1.45	6.88	1.70	
EHA[G]	4.01	1.23	4.01	0.69	3.98	1.15	
EHA[NR]	7.34	1.11	7.10	0.63	6.88	0.97	
EHA[G+NR]	4.01	1.04	4.01	0.60	3.98	0.95	
EHA[G+B]	4.01	1.14	4.01	0.73	3.98	0.89	

Road Networks – 10000 Random $(s, t)$ pairs							
Graph	U.S. NHPN		Florida		Alabama		
#GCP	110		51		74		
Phase	Time	#del_min	Time	#del_min	Time	#del_min	
pruning	0.92	–	0.43	–	0.74	–	
gc[MZ]	5.64	201156	1.46	37284	1.34	32940	
gc[new]	4.68	144848	0.94	30889	1.30	29620	

further improvements and makes EHA[G+NR] along with EHA[G+B] to be the fastest implementations of the exact algorithm. Regarding pruning and gap closing (phases 2 and 3 of the exact algorithm), our experiments revealed that these two phases were performed only for a very small number of instances, actually less than 2%; see lower part of Table 1. This implies that for the 98% of the instances tested, the time bounds of Table 1 (upper part) essentially provide the total running time of the exact algorithm. We made two implementations of gap

closing: the first is the one described in [9] (gc[MZ]), while the second is our method described in Section 4.2 (gc[new]) that uses Pareto path information. The larger the graph, the better the performance of gc[new], since much more paths are discarded and thus less delete-min operations are executed.

Regarding the experiments on real road networks from USA, we consider the high-detail state road networks of Florida and Alabama, and the NHPN (National Highway Planning Network) covering the entire continental US. The data sets of the state networks consist of four levels of road networks (interstate highways, principal arterial roads, major arterial roads, and rural minor arterial roads). The Florida high-detail road network consists of 50109 nodes and 133134 edges, the Alabama high-detail road network consists of 66082 nodes and 185986 edges, and the US NHPN consists of 75417 nodes and 205998 edges. These particular data sets were also used in [14]. In all cases the cost and the resource of an edge were taken equal to the actual distance of its endpoints (provided with the data) multiplied by a uniform random variable in  $[1, 2]$  (to differentiate the cost from the resource value), and subsequently scaled and rounded appropriately so that the resulting numbers are integers in  $[0, 10000]$ . The non-linear objective function considered in all cases was the same as that for grid graphs and DEMs. The experimental results (see Table 2) are similar to those obtained using grid graphs and DEMs.

## References

1. R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows*, Prentice-Hall, 1993.
2. J. Beasley and N. Christofides, "An Algorithm for the Resource Constrained Shortest Path Problem", *Networks* 19 (1989), pp. 379-394.
3. E.W. Cheney, *Introduction to Approximation Theory*, McGraw-Hill, NY, 1966.
4. S. Gabriel and D. Bernstein, "The Traffic Equilibrium Problem with Nonadditive Path Costs", *Transportation Science* 31:4(1997), pp. 337-348.
5. S. Gabriel and D. Bernstein, "Nonadditive Shortest Paths: Subproblems in Multi-Agent Competitive Network Models", *Comp. & Math. Organiz. Theory* 6 (2000).
6. G. Handler and I. Zang, "A Dual Algorithm for the Constrained Shortest Path Problem", *Networks* 10(1980), pp. 293-310.
7. M. Henig, "The Shortest Path Problem with Two Objective Functions", *European Journal of Operational Research* 25(1985), pp. 281-291.
8. D. Hensen and T. Truong, "Valuation of Travel Times Savings", *Journal of Transport Economics and Policy* (1985), pp. 237-260.
9. K. Mehlhorn and M. Ziegelmann, "Resource Constrained Shortest Paths", in *Algorithms – ESA 2000*, LNCS 1879 (Springer-Verlag 2000), pp. 326-337.
10. P. Mirchandani and M. Wiecek, "Routing with Nonlinear Multiattribute Cost Functions", *Applied Mathematics and Computation* 54(1993), pp. 215-239.
11. C. Papadimitriou and M. Yannakakis, "On the Approximability of Trade-offs and Optimal Access of Web Sources", in *Proc. 41st FOCS 2000*, pp. 86-92.
12. K. Scott and D. Bernstein, "Solving a Best Path Problem when the Value of Time Function is Nonlinear", preprint 980976 of the *Transport. Research Board*, 1997.
13. G. Tsaggouris and C. Zaroliagis, "Non-Additive Shortest Paths", Tech. Report TR-2004/03/01, Computer Technology Institute, Patras, March 2004.
14. F.B. Zhan and C.E. Noon, "Shortest Path Algorithms: An Evaluation using Real Road Networks", *Transportation Science* 32:1(1998), pp. 65-73.

# Author Index

- Adler, Micah 496  
Agarwal, Pankaj K. 4  
Amir, Amihoud 16  
Anshelevich, Elliot 28  
Arge, Lars 4, 40  
August, David I. 677  
Azar, Yossi 53, 65
- Bansal, Vikas 77  
Bartal, Yair 89  
Becchetti, Luca 98  
Beier, Rene 616  
Bhargava, Ankur 110  
Bodlaender, Hans L. 628  
Boros, Endre 122  
Busch, Costas 134
- Carroll, Douglas E. 146  
Chan, Hubert 157  
Chen, Ning 169  
Cheriyān, Joseph 180  
Chlebík, Miroslav 192  
Chlebíková, Janka 192  
Chrobak, Marek 204  
Chwa, Kyung-Yong 484  
Cohen, Rami 216  
Cohen, Reuven 228  
Cole, Richard 240  
Czygrinow, Andrzej 252
- Damerow, Valentina 264  
Deng, Xiaotie 169  
Díaz, Josep 275
- Eisenberg, Estrella 16  
Elbassioni, Khaled 122  
Elsässer, Robert 640  
Emiris, Ioannis Z. 652  
Epstein, Leah 287  
Erickson, Jeff 4
- Fekete, Zsolt 299  
Fellows, Michael R. 1, 311  
Fischer, Simon 323  
Fleischer, Rudolf 335  
Fogel, Efi 664
- Fotakis, Dimitris 347  
Fusco, Giordano 772
- Gagie, Travis 359  
Garg, Naveen 371  
Georgiadis, Loukas 677  
Gidenstam, Anders 736  
Goel, Ashish 146  
Golubchik, Leana 689  
Gonen, Rica 383  
Gurvich, Vladimir 122
- Ha, Phuong H. 736  
Halperin, Dan 664  
Hańćkowiak, Michał 252  
Hassin, Refael 395, 403  
Hazay, Carmit 414  
Henzinger, Monika 3
- Jawor, Wojciech 204  
Jordán, Tibor 299
- Kamphans, Tom 335  
Kandathil, David C. 240  
Kettner, Lutz 702  
Khandekar, Rohit 371  
Khuller, Samir 689  
Kim, Pok-Son 714  
Kim, Yoo-Ah 689  
Klein, Rolf 335  
Knauer, C. 311  
Kosaraju, S. Rao 110  
Koster, Arie M.C.A. 628  
Kovaleva, Sofia 426  
Kowalik, Lukasz 436  
Kreveld, Marc van 448, 724  
Kuhn, Fabian 460  
Kulkarni, Raghav 472  
Kutzner, Arne 714
- Langetepe, Elmar 335  
Larsson, Andreas 736  
Lee, Jae-Ha 484  
Levin, Asaf 395  
Lewenstein, Moshe 414  
Litichevsky, Arik 53

- Liu, Junning 496  
 Lorenz, Ulf 749  
  
 Magdon-Ismaïl, Malik 134  
 Mahajan, Meena 472  
 Malhotra, Varun S. 508  
 Martens, Maren 520  
 Mavronicolas, Marios 134  
 Mecke, Steffen 760  
 Mehlhorn, Kurt 702  
 Meyer auf der Heide, Friedhelm 77  
 Monien, Burkhard 640  
 Moscibroda, Thomas 460  
 Mucha, Marcin 532  
  
 Nishimura, N. 311  
 Nüsken, Michael 544  
  
 Pagh, Anna 556  
 Pagh, Rasmus 556  
 Papatriantafyllou, Marina 736  
 Park, Sang-Min 484  
 Peleg, David 228  
 Pellegrini, Marco 772  
 Pion, Sylvain 702  
 Porat, Ely 16  
  
 Ragde, P. 311  
 Rawitz, Dror 216  
 Raz, Danny 216  
 Rémila, Eric 568  
 Richter, Yossi 65  
 Roditty, Liam 580  
 Rosamond, F. 311  
 Rubinstein, Shlomi 403  
 Ružić, Milan 592  
  
 Salavatipour, Mohammad R. 180  
 Samoladas, Vasilis 40  
 Sanders, Peter 784  
 Sankowski, Piotr 532  
 Schamberger, Stefan 640  
 Schirra, Stefan 702  
 Serna, Maria 275  
 Sgall, Jiří 204  
 Shargorodskaya, Svetlana 689  
 Sigurd, Mikkel 797  
  
 Skutella, Martin 520  
 Sohler, Christian 77, 264  
 Sokol, Dina 414  
 Speckmann, Bettina 724  
 Spieksma, Frits C.R. 426  
 Spirakis, Paul 134  
 Stappen, A. Frank van der 448  
 Stee, Rob van 287  
 Stege, U. 311  
 Sun, Xiaoming 169  
 Szymańska, Edyta 252  
  
 Tănase, Mirela 809  
 Tarjan, Robert E. 677  
 Thilikos, Dimitrios M. 275, 311  
 Thorup, Mikkel 556  
 Tichý, Tomáš 204  
 Triantafyllis, Spyridon 677  
 Trippen, Gerhard 335  
 Tsaggouris, George 822  
 Tsigaridas, Elias P. 652  
 Tsigas, Philippas 736  
  
 Veltkamp, Remco C. 809  
 Vöcking, Berthold 323, 616  
  
 Wagner, Dorothea 760  
 Wan, Yung-Chun (Justin) 689  
 Wattenhofer, Roger 460  
 Wein, Ron 664  
 Werneck, Renato F. 677  
 Whiteley, Walter 299  
 Whitesides, S. 311  
 Winkel, Sebastian 784  
 Wolle, Thomas 628  
  
 Yao, Andrew Chi-Chih 169  
 Yap, Chee 702  
 Yi, Ke 40  
 Yu, Hai 4  
 Yuster, Raphael 604  
  
 Zachariasen, Martin 797  
 Zaroliagis, Christos 822  
 Zhang, Lisa 28  
 Ziegler, Martin 544  
 Zwick, Uri 580, 604